AN INTRODUCTION TO TRS-80 ASSEMBLY LANGUAGE PROGRAMMING

A System for
Efficient Independent Study

For TRS-80 Models I and III

REMASSEM
SECOND EDITION

presented by

REMSOFT, INC.
571 East 185th Street
Euclid, Ohio 44119
(216) 531-1338

® TRS-80 IS A REGISTERED TRADEMARK OF TANDY CORPORATION.

The programs and program listings contained and discussed in this course are provided for tutorial purposes only. The purchaser, and only the purchaser, is authorized to incorporate the subject program segments into his/her own programming, but REMSOFT, INC., shall have no liability or responsibility to purchaser or any other person on entity as far as liability, loss, or damage caused or alleged to be caused directly or indirectly by use of such programs or program segments which are part of this or any other package provided by REMSOFT, INC., including but not limited to any loss of business or anticipated profits arising from such loss of business or any interruption of service, or consequential damages resulting from any use of such programs or program segments.

OVERVIEW OF THE REMASSEM COURSE

The **REMASSEM** course was developed and recorded by me (Joseph E. Willis) for the novice assembly language student who wishes to study individually and who has ready access to either a Model I or a Model III TRS-80. The present version of **REMASSEM** is the second edition for the Model I and the first version for the Model III. Tape-based and diskette-based packages, differing only in the physical media provided, are available for either computer model.

REMASSEM successfully combines several principles of learning into an effective, moderately priced package for learning assembly language by independent study. As you listen to the audio lectures, positive reinforcement for what is being heard is gained by a series of video images brought up by you upon audible cue. Thus, there is positive interaction facilitating comprehension and retention.

SUMMARY OF THE FEATURES OF THE COURSE:

* Ten lectures, averaging more than 40 minutes each, on high-quality audio tapes.

* A driver program (BLKBRD) to make the TRS-80 video monitor serve as a "blackboard" for me.

* Display programs for each lecture to provide illustration of pertinent points being discussed in the lecture. These are displayed by BLKBRD.

* Step-by-step dissection of complete, stand-alone assembly language programs chosen to be useful in your own programming, as well as to serve as how-to guides for you to study.

* Information on how to access and use several powerful ROM routines in your own programs.

* A study manual which provides necessary information for efficient study along with documentation materials. A highlight is an extensive description and discussion of a game program, to illustrate relatively advanced assembly language programming techniques.

REMASSEM
TABLE OF CONTENTS

CHAPTER 1
GETTING STARTED

Welcome to the wonderful world of assembly language programming for the TRS-80 Models I and III computers! This chapter will tell you what you need to know to begin study of the course. A suggested study sequence will be given, along with suggestions regarding additional reference materials and software you may wish to acquire to aid in your pursuit of proficiency in assembly language programming.

MATERIALS PROVIDED WITH THE COURSE

The REMASSEM course materials are housed in an attractive looseleaf binder which also has storage capacity for eight tape cassettes. Included are five C-90 audio cassettes which provide the ten audio lectures. If you have the tape-based version, you will find that the other three cassette slots contain the BLKBRD tape and display tapes which will be used with the audio lectures. There are two copies of each of these digital files on the tapes.

If you have the diskette-based version, you will find a diskette inserted in a plastic sleeve behind this booklet. This contains the BLKBRD program and the display files it uses.

The registration card should be filled out and mailed back to REMsoft at your earliest convenience. This is very important, since you must be a registered owner of REMASSEM before any defective tapes, diskettes, etc., will be replaced. We also may need to send out update information at some time in the future; for this, we will use the list of registered owners.

You should find a couple of "question" sheets inserted into this manual. These are to be used when you have a question pertinent to your study which is not answered anywhere within the course. These questions must be strictly within the scope of the topics covered in REMASSEM! The world of assembly language programming is entirely too large to allow time to answer just ANY question.

The final item provided with REMASSEM is the study manual you are now reading. If any item is missing, please contact the source from whom you purchased the course for prompt replacement. NOTE -- Disregard this notice if this manual is missing, since you cannot read it anyway! Seriously, I want you to have all of the materials available when you start your study, so check your package now.

LOADING INSTRUCTIONS

This section is split into two portions, since the loading procedure is necessarily different for tape-based and disk-based systems. Select the appropriate section and read it thoroughly, so we can get started on your study.

TAPE SYSTEMS

There are three kinds of cassette tapes included in the course. The first tape of interest is the BLKBRD tape which you must load before each session. This is a SYSTEM tape to be loaded in the usual manner by typing "SYSTEM", typing "BLKBRD" when the prompt appears, and entering "/" after the blackboard program has loaded and the prompt re-appears. (Of course, you know that the quotes enclosing the commands shown above are not actually typed.) Remember to press the <ENTER> key after each of the commands. The BLKBRD program loads any of the display tapes for the audio lectures and controls their display. All of the computer-readable tapes are recorded at the standard 500 baud rate, so if you have a Model III, you must select the low cassette rate.

The next tape you should load is the appropriate display file for the session of interest. There are two sessions in each file -- in fact, there is a duplicate file following the first on each side of each cassette. I have heard a rumor that one copy of a program is never enough! Right??

The third type of tape is the audio lecture for the session of interest. If you use the same recorder that you used to load the other tapes into your TRS-80, be sure to unplug the recorder cables so you can hear the lecture.

DISK SYSTEMS

REMassem diskette has its own backup routine!

Power up your TRS-80 in your normal manner. Place the REMASSEM diskette in drive 0 and press the <RESET> button. The BLKBRD program will be automatically brought into memory and a menu will be displayed. When you select the desired session and press the appropriate key, that display file will be loaded and the first display image will be shown.

It has often been said, "Watch that first step, it's a BIGGIE!" This has never been more true than now. The first step is for you to make at least one complete backup of the REMASSEM diskette. The next step, equally important, is to store the original diskette in a safe place away from all computer gremlins. After using diskettes for

several years, I am convinced that there are few things in God's universe more fragile than a unique diskette! Things seem to go along all right if you have a backup copy. An intelligent person like you does not need a second prompting, right? GOOD!

BLKBRD COMMANDS

Here are the commands you will find useful to control what is shown on the screen as you listen to the audio tapes:

back to beginning of 1st of 2 audio sessions

<B>

Each display file contains screen images for two audio sessions. The <B> key is used to return to the beginning of the first session for the display file you have memory-resident. An easy way to start over if you find your attention wandering!

to next screen

<DOWN ARROW>

Depress the down arrow key to tell your TRS-80 to display the next screen image. To keep what you are seeing in synchronization with what you are hearing, press <DOWN ARROW> each time you hear the "beep" on the audio tape.

session to menu

<M>

Used whenever you want to bring up a menu of the major topics of discussion in the display file which is currently memory-resident. One use for the <M> key is to allow quick positioning to the start of the second session in each display file. It is also extremely useful when you are resuming your study "mid-lesson" after some hiatus. This function is also useful when you realize that what you are looking at on the screen bears only faint resemblance to what I am saying in the lecture -- this may be either your computer's fault or my fault. We both know it could not possibly be YOUR fault, don't we? When and if this occurs, stop the audio tape, depress the <M> key, select the appropriate subject, and GO, GO, GO! For your convenience, Appendix A of this manual contains a list of the menus.

to Main Menu

<F> or <X>

Use either one whenever you wish to load a different display file into memory. This saves you from having to reload the BLKBRD routine each time you want to change to a new session.

SUGGESTED STUDY SEQUENCE

This course is an integrated package of written and audio-visual information which can help you learn assembly language programming on an individual basis, at your own pace. As such, there is much flexibility built into REMASSEM. Although you certainly may "pick and choose" the topics you are most interested in, I feel that your progress will be much faster, with less effort, if you adhere to the following course of study.

1. Read and study Chapter 1 of the manual. Then scan Chapter 2 for just the important concepts. Delay thorough study of Chapter 2 until step 3 below.

2. Thoroughly digest audio lessons 1-5.

3. Study Chapter 2 of the manual and use your assembler to work through all examples.

4. Listen to audio lessons 6 and 7, using Chapter 3 in the manual as suggested in the lectures.

5. Listen to audio lessons 8 and 9, using Chapter 4 in the manual as suggested.

6. Listen to audio lecture 10.

7. Study thoroughly Chapter 5 in the manual, keying in the program and making suggested modifications, etc.

8. Repeat steps 1 through 7 until the desired skill level is reached!

COPYRIGHTS, TRADEMARKS, ETC.

This documentation was compiled with the help of a TRS-80 and a word-processor program, which is indeed an excellent use of a computer. Without computer assistance, it is likely that this booklet would be much smaller than it is. However, there is a problem!

All those fine folks at Tandy, Zilog, Apparat, and yes -- even REMsoft, insist that we acknowledge all of their trademarks, copyrights, etc. A trademarked item is commonly designated by displaying a registration mark which is a superscripted 'R' with a circle around it, right after the trademarked symbol, word, or phrase. A copyright symbol is typically a 'C' within a circle.

Both are very difficult, if not entirely infeasible, to accomplish with an ordinary computer printer, even with an excellent word processor like SCRIPSIT. So here is what I have done -- I have put this section at the front of this booklet to list all of the trademarked, copyrighted, etc., terms together below. Since you are an outstanding, perceptive, very selective, and discerning person (you proved all that by purchasing REMASSEM!), I know you will note each one and imbed it in your memory. Forever after, when you see the trademarked and/or copyrighted terms used, you will immediately remember to whom it should be attributed. Whew, that's a relief! That surely does keep me out of trouble with the companies below:

FIRM	COPYRIGHT or TRADEMARK
Apparat, Inc. 4401 Tamarac Pkwy. Denver, CO 80237	NEWDOS, NEWDOS+, NEWDOS80, SUPERZAP
Logical Systems, Inc. Mequon, WI 53092	LDOS
Microsoft Consumer Prods 400 108th Ave, NE Bellevue, WA 98004	M80, L80, EDIT, Editor/Assembler-Plus
MISOSYS 5904 Edgehill Dr. Alexandria, VA 22303	DISKMOD, EDAS 3.5, DSMBLR
REMsoft, Inc. 571 E. 185 Euclid, OH 44119	REMASSEM, REMDISK-1
Soft Sector Marketing, Inc. 6250 Middlebelt Road Garden City, MI 48135	MOD III ROM COMMENTED
Tandy Corporation 700 One Tandy Center Fort Worth, TX 76102	Radio Shack, EDTASM, SCRIPSIT, TRS-80
Zilog 10460 Bubb Road Cupertino, CA 95014	Z80, Zilog

Speaking of copyrighted material, the entire REMASSEM package is copyrighted. However, with all the restrictions, limits, exemptions, etc., listed on the inside cover of the booklet, I encourage you to use, freely and frequently, the routines and techniques described herein in your own programming efforts. I DO ask, if you include any

of the routines in a program you offer for sale, that you state "PORTIONS COPYRIGHTED BY REMsoft, INC." in some prominent place on the package. If you have any spare money lying around you can send that to us, too; but we are not asking for money, JUST PROPER CREDIT. Fair enough?

PERTINENT REFERENCE MATERIALS

To get maximum benefit from this course, I suggest that you obtain and study the following:

A. EDITOR/ASSEMBLER

Although I think REMASSEM is a good "stand-alone" course, it can only talk about assembly language programming. You are the one who will have to DO something about assembly language programming. It is up to you, the student, to develop the desired skill by a combination of study and practice. You cannot learn assembly language programming simply by observation -- you must participate!

The first thing you need in order to participate is an editor/assembler program designed for your computer. I give a long recitation in Chapter 2 about assemblers and their characteristics, so I will not say very much here except that assemblers fall into two main categories:

1. Ordinary (non-macro) assemblers

2. Macro assemblers.

I now want to mention several of the commercial assembler packages available at the time of this writing (mid 1982). I have not had extensive personal experience with all of the packages mentioned, but have had enough comments from friends, students, etc., to be able to provide some perspective for you. Do not consider this list exhaustive; the opinions expressed are largely my own, so blame me if you disagree. Let me know your opinions and experiences with the assembler you choose.

Ordinary Assemblers

"Series I Editor/Assembler" by Radio Shack is available for both Models I and III in either cassette or disk versions. It is a very good ordinary assembler, although it has no particularly outstanding features in my opinion. On the good side, it is easy to use and has no bugs as far as I know. It is relatively inexpensive and very readily available. I have chosen this assembler for the examples I show in REMASSEM, since it is a typical assembler and should be easily obtainable by anyone having a TRS-80 computer.

"Editor/Assembler-Plus" by Microsoft is available for Models I and III tape- and disk-based systems. I have not used this package personally, but have received good comments from students. It now also has macro capability (in the disk-based version only), so perhaps I should include it under the macro heading as well. Looks good! An outstanding feature is a built-in debug facility. Maybe you are better at it than I, but I need all the help I can get! The only way I can get my computer to do what I want it to do is to make sure that is also what I have told it to do!

"EDAS 3.5" by MISOSYS is a very sophisticated editor and assembler for both Models I and III, disk systems only. Although it does not support macros, and is thus considered an "ordinary" assembler, it has many of the powerful features of a macro assembler. I am impressed with its list of features, and recommend this one if you have a disk system.

"NEWDOS EDTASM" There is also an editor/assembler contained in the various versions of the NEWDOS disk operating system by Apparat. This assembler is based on the original Radio Shack tape-based EDTASM. No documentation is given except for the enhancements; Apparat states that you must separately purchase the EDTASM from Radio Shack to get the needed documentation and avoid violation of the copyright laws. An outstanding feature of this assembler is that it provides a sorted symbol cross-reference listing. This is very useful with large assembly language programs.

"DISKMOD" by MISOSYS is for the Model I only, and is specifically designed to upgrade the original tape-based EDTASM (from Radio Shack) to a disk-based version, with enhancement of some of the features. If and only if you have a Model I disk system and the original EDTASM will you need (or indeed can you use) this package. If you meet these specs, I suggest you get DISKMOD. I have used this extensively for years and particularly like the sorted symbol table listing and the capability of moving source lines around in the program. Both of these features help you keep the source code organized.

Macro Assemblers

Radio Shack offers a macro assembler for Model I disk only. The package was developed by Microsoft and appears identical to that offered for some time under the Microsoft name. It appears that Radio Shack will offer a macro assembler for both Models I and III as part of their announced "assembly language development system", but I have not seen it yet. Perhaps by the time you are reading this, it will be available.

As I mentioned above, the Editor/Assembler-Plus package (disk version only) by Microsoft has macro capability.

General Comments

I recommend that your initial assembler be one of the ordinary assemblers, rather than a macro assembler. Although a macro assembler is much more versatile and may be your choice later, it is much more complicated to use. In fact, it likely will hinder your progress in learning assembly language by bogging you down in the details of operation of the assembler program itself. However, the choice is yours.

If you already have one of the macro assemblers, give it a good try before you run out to spend any more hard-earned money. (If your money was not hard-earned, it's not so bad, so spend away! A joke -- just a joke!) You may find that the macro assembler is not so difficult to use after all, and you will certainly like the features which will allow you to assemble, easily, very large source programs later on. This way, you only have to learn one assembler.

Whatever assembler you choose, it is imperative that you purchase one and practice using it. I will be saying that again and again simply because it is true!

B. EXTRA READING

REMASSEM is the best way to learn TRS-80 assembly language programming (completely unbiased opinion, of course!). However, at some point in your study, you will feel the need for additional reference materials, so I want to list some with which I am familiar. As usual, I give my opinions. Let me start with some books.

Barden, TRS-80 Assembly-Language Programming, Radio Shack.

This book was included in the original REMASSEM as a textbook. It is good in that it is specific to the TRS-80, and is inexpensive enough that you probably will want to get it. We no longer include it as a part of REMASSEM for two reasons:

1. You probably already have it, and there's no need to increase the price of REMASSEM to cover the expense of something you already have.

2. It is more suited for the Model I than it is for the Model III.

Leventhal, Z-80 Assembly Language Programming, Osborne McGraw-Hill.

This is a very complete book. I think the neophyte programmer could well be overwhelmed by the amount of material given in the book if he/she attempts to use it as a text for self-teaching, but the great amount of information given makes this an excellent reference book. It should be on your bookshelf.

Barden, The Z-80 Microcomputer Handbook, Sams.

This book is more thorough than the other one above by Barden, in that the details of the Z80 chip are discussed as well. I prefer Leventhal for my money, but (as with any other book in this list) it can be useful at times and, if money is no object, should be readily at hand.

Zaks, Programming the Z80, Sybex

This book is rather bulky, but upon inspection it is seen that a lot of the bulk is derived from spacious representations of the instruction set. I do not think it has the depth of information that several of the others provide, and I quarrel with some of the statements made, but you may find Dr. Zaks' book to your liking. I personally can live without it.

Spracklen, Z-80 and 8080 Assembly Language Programming, Hayden.

This is perhaps my favorite as a study book, at least as far as the approach that Mrs. Spracklen takes toward teaching assembly language. You can tell that I admire this approach if you have seen the book, because the manner in which I have developed the topics in the first couple of lectures parallels closely the technique she follows. I think this book can be somewhat confusing for the neophyte, in that both Z80 and 8080 mnemonics are given. (I'll discuss what "mnemonics" are in lecture 1.) Eventually you will want to know what the 8080 mnemonic is for a given Z80 instruction (if there is an 8080 instruction for the Z80 instruction), and at that point perhaps the Spracklen book will be quite useful to you.

Howe, TRS-80 Assembly Language, Prentice-Hall.

This is a useful but somewhat overpriced book, in my opinion. The information given is different in many cases to what the other books provide and therefore may deserve a spot on your bookshelf, but examine it closely first. Several of the areas Howe discusses are specific to the Model I and are not directly applicable to a Model III.

C. OTHER REFERENCE MATERIALS

You must realize that the list could go on forever, but I want to give you some descriptions of helpful products available from several sources. The study of any computer language is an iterative process -- you study a while, try programming for a while, study some more, etc., gradually becoming proficient. One of the best ways to study (after you have learned the instruction set) is to examine other programmers' work to see the techniques they have employed. Most programmers are not as foolhardy as I am, preferring not to display their source listings for the whole world to belittle! So, we must get crafty and use a "disassembler" to examine object code. As you may guess, a disassembler at least partially reverses the assembly process. An assembler takes source code and generates object code -- a disassembler takes object code and generates some semblance of source code. (I realize that is quite a lot of jargon! These terms will be more meaningful to you after you read the first part of Chapter 2.)

Of course, it is not as easy as all that, since the disassembler has no way of figuring out a meaningful label for a given address reference. So the disassembly listing is not as easy to understand as the original, true source listing, but still very useful for study. Actually, I have seen some programs where the disassembled listing WAS as easy to understand as the source listing, but that is another story!

The DSMBLR program by MISOSYS is a very good disassembler program to have. It will even generate a "source" file on tape or disk for input to your assembler. The NEWDOS disk operating system also provides a disassembler which has many good features. I normally use these two disassemblers in combination. I like some features from one, and other features from the other -- together, they make an excellent team!

In addition to disassemblers, you may be curious about the ROM routines in your computer. After all, many of the functions that BASIC provides are equally useful in your assembly language programming. There are several books which "open up" this treasure for you to study and use. For the Model I, a good reference to have is "The Book -- Volume I and II" by Insiders Software Consultants, Inc. Another good one is "TRS-80 Super Map" by Roger Fuller.

Also for the Model I are the series of "Other Mysteries" books published by IJG Computer Services. Particularly useful are "Microsoft Basic Decoded & Other Mysteries", by James Farvour and "The Custom TRS-80 & Other Mysteries", by Dennis Bathory Kitsz. For the Model III, a good reference is "MOD III ROM COMMENTED" by Soft Sector Marketing, Inc. Although this does not tell you how to use any of the routines, it is about as complete a disassembly of the ROM as could be listed without violating Radio Shack's copyright! Careful study will prove worthwhile.

Do not forget the appropriate Service and Technical Manuals for your TRS-80. These are available from Radio Shack, although apparently this is not widely known even among Radio Shack managers, and should be obtained. They contain much material which is very useful.

WHAT TO DO NEXT

I suggest you scan Chapter 2, just to begin to become familiar with the topics discussed, before proceeding with audio lessons 1 through 5. There is a lot of stuff you must cover to learn assembly language, with a lot of jargon, so do not be upset if you don't understand everything completely the first time through.

CHAPTER 2
THE CAT CHASED THE MOUSE

There are two goals for this chapter:

1) to give you information about what an assembler is and what it does for us.

2) to have you learn the mechanics of using your particular assembler by playing with a trivial, short program.

It has been my experience that many students, particularly when they are studying independently (as you probably are), attempt to learn assembly language by simply listening and looking! You would not think of learning BASIC by reading and studying only and yet you (perhaps) expect to learn the much more difficult assembly language in this manner.

You may succeed, but if you do you are well above most students in capability! I have found that much practice is usually necessary before one becomes comfortable in any "foreign" language, computer or otherwise. Particularly if you want a writing knowledge of assembly language as contrasted with just a reading knowledge, you must be prepared to spend many hours in practice. In order to practice efficiently, you must know how to use the particular assembler you own. You DO have an assembler program, don't you? Let's take some time now to discuss assemblers.

Quite simply, an assembler is a machine language program which will convert an assembly language program into the corresponding machine language program. The assembly language program is called the "source" code, and the machine language program it generates is called the "object" code.

The computer must work at all times with machine code (object code) -- that is all it understands! When we think we are running a BASIC program, we are really running a machine language BASIC interpreter which, logically enough, interprets our BASIC source program a portion at a time and executes appropriate machine language routines stored in ROM as it performs the interpretation. Note that I have called the original code "source code", whether it was in assembly language or BASIC. This is the common jargon -- a program in any language that the computer cannot execute directly without translation or interpretation is called source code.

Whereas the program that executes BASIC source code is called an interpreter since it must continually and repetitively interpret the source program each time it executes it, the assembler is a form of compiler. What is a compiler? Normally a compiler translates the source code directly into object code; that is, the output from the compiler is machine language, ready for execution by the computer as soon as it is loaded into memory. A FORTRAN compiler normally is of this type. So are most ordinary assemblers. Radio Shack now sells a BASIC compiler, too, which generates object code.

There are exceptions to this, however, wherein some sort of intermediate code is produced which requires a run-time module present during execution to perform the interpretation of the intermediate code. Pascal and CBASIC are examples of this type of compiler.

So, now that we know that an assembler is a type of compiler, let us look in more detail into what an assembler is. There are two main types of assemblers:

1) ordinary assembler.

2) macro assembler.

They differ largely in the way they translate source code statements.

An ordinary assembler takes each line of source code you have written and translates it, instruction by instruction, into the corresponding object code. Since many assemblers allow only one instruction per line of source code, there is a direct, one-to-one correlation between source instructions and object instructions. Furthermore, each statement must be a valid instruction; in particular, the exact mnemonic representation must be followed. There is certainly a place for creativity in computer programming, but this is not the place!

If the assembler does not recognize the statement as a valid instruction, it will let you know when it assembles your source program. You must then correct the syntax error(s) and try again.

Actually, most of what you have just read applies as well to a macro assembler. Every statement must be recognized as valid or you will get an error message! Neither assembler is a mind reader -- that costs extra, much more! The main difference is that, with a macro assembler, you can make up new instructions and use them in your programming, as long as the new instructions are built up from valid, ordinary instruction sequences AND as long as you

tell the assembler the "definitions" of these new instructions. At its best, you can use a macro assembler to allow you to use a sort of shorthand to represent longer sections of code.

It is important to realize that the object code will be the same whether you use macros or whether you spell out each instruction explicitly each time. Macros only shorten the amount of keying in you must do to get the source code.

There is another difference between ordinary assemblers and macro assemblers, one which is perhaps even more important than that above. An ordinary assembler generates "absolute" code, while a macro assembler usually generates "relocatable" code. What in the world does Willis mean by this?

Object code normally will execute properly only when it is placed in the memory locations where it was assembled to execute. This may be intuitively obvious to you, but it is not for many people, so let us talk about it a little bit. Scanning the Z80 instructions in Appendix B will reveal many instructions which do not involve specific areas of memory, for example, the register-to-register loads, the Boolean operations involving only registers, and several others. Quite a few instructions, however, require reference to specific memory locations. If these references are to labelled instructions in your program itself and you move the object code to a different area of memory, the program will bomb! How about an example?

Let's say that you had the following section of code in one of your programs:

```
          ORG  7300H
LOOP      LD   A,(DATA)

          .
          .
          .

          JP   LOOP
          ORG  7380H
DATA      DEFS 1
```

The periods indicate a sequence of instructions not pertinent to this discussion. In the instruction labelled LOOP, we are fetching a byte of data stored in the memory location labelled DATA and placing it in the accumulator. In the instruction just past the column of periods we see JP LOOP, which jumps to the memory location we have called LOOP.

FOR BOTH INSTRUCTIONS, THE CODE GENERATED BY OUR ORDINARY ASSEMBLER WILL SHOW SPECIFIC, ABSOLUTE MEMORY LOCATIONS. That is, if we specify at the time of assembly that LOOP is at 7300H and DATA is at 7380H, and then we later move the object code from 7300-7380H to 7400-7480H, the first instruction will still fetch the byte at 7380H, not 7480H, and place it in the accumulator. The JP LOOP instruction will be a jump to 7300H, not our desired 7400H destination! We must reassemble the source code with new specifications to the assembler in order to have the program work properly at 7400H.

How does the macro assembler differ from this? A macro assembler generates a type of intermediate code which does not contain absolute addresses. Before this intermediate code (which is often called "relocatable" code) can be executed, it must be converted by a linker program to standard object code. Here is an important point to realize about the object program after the linker program has done its job -- the code is indistinguishable from that produced by an ordinary assembler! All references which were absolute addresses with the ordinary assembler are absolute addresses now. The final object code is no more "relocatable" than it was in the example above!

So why call it relocatable code? Well, it is relocatable insofar as you do not have to reassemble to generate an object program to run at 7400H. In fact, you normally do not specify any address at which the program is to run at assembly time with a macro assembler. This is done at linking time. To get an object program which will execute properly at 7400H, you simply relink, specifying 7400H as the new origin. (This assumes that the program worked right at 7300H -- there's no magic, I'm sorry to say!)

The advantage of this approach is that you can devise very useful subroutines which you will use many times in your future programming and assemble them into these relocatable modules, forming a so-called "library". Then, when you need one or more of them in a program you are just writing, you can specify to the assembler that the modules are "external" to the new program. That is, these routines are not present in this module. If you fail to tell the assembler this, it will issue appropriate error messages.

The assembler will generate code which will allow the linker to resolve the external references at linkage time. Obviously, you must tell the linker where to find these library routines; once again, it cannot or at least will not read your mind. Another advantage of a macro assembler is that the source code for the entire program does not need to fit into available memory at the same time. Each module can be assembled separately and then linked later. At the risk of being absurd, I will remind you that the final object code must fit available memory.

It is apparent that a macro assembler is much more flexible than an ordinary assembler. So why not use one all the time? Why does Willis recommend that you start off with an ordinary assembler?

The goal of studying assembly language is to learn assembly language! It is almost a law that a more versatile system is a more complex system and the macro assembler which Radio Shack offers for the Models I and III is a very complex system to learn. This may delay your study of assembly language itself. I hope what I have said in the last several paragraphs helps you if you already have this assembler, but I personally feel that you would progress much faster if the assembler were simple enough to use to permit you to focus on the goal of learning assembly language as quickly as possible. The Radio Shack macro assembler is a marvelous piece of software which you may want to use in the years to come, after you gain some experience.

Assembler Pseudo-Operations and Assorted Esoterica

The assembler normally needs to know several things before it can do an acceptable job of assembling our masterpiece of a program. For this discussion, I will describe what many of these things are, using the Radio Shack Series I Editor/Assembler as a typical ordinary assembler. The topics will be discussed in the following order:

ORG
EQU
DEFL
END
DEFB
DEFW
DEFM
DEFS
Miscellaneous Loose Ends

It should be emphasized once more that you must read the documentation which came with your assembler. The mnemonic for a given command which I will be discussing may be spelled differently even though its function is exactly the same. Your assembler will be nice to you if and only if you play by its rules -- you will save a lot of "grief" if you learn those rules now! So the examples I show may not work verbatim with your assembler, but the discussion should still be quite useful in helping you to understand your particular assembler.

ORG Pseudo-Op

As I said earlier, an ordinary assembler assembles a program to execute in one specific memory area. The ORG ("origin") pseudo-op is how we tell the assembler where the desired memory starts. If you leave this out, most assemblers will assume a starting address of 00. This does not work well at all for the TRS-80, since there is ROM memory at that address; there is no way you will be able to load the object code into that area for execution! So we must have an ORG statement if we are using an assembler like the Series I.

If you have a disk system, the lowest memory address you can use as an origin is 5200H unless you like the excitement of guessing what vital portion of the disk operating system you zapped when you loaded your program! In fact, if you have a Model III, I suggest you use 5600H, so you can use the DEBUG utility. DEBUG will overwrite at least part of your program otherwise.

For a tape system, the lowest address is 42E9H. Tape-based Model III programs seem to work better when I use 4400H as origin, but maybe that is only my imagination. The Radio Shack documentation says 42E9H.

You DO NOT USE the ORG pseudo-op in the normal situation with a macro assembler. The desired origin is specified when using the LINKER, not when using the ASSEMBLER.

EQU Pseudo-Op

This will be one of your most frequently used instructions to the assembler. The EQU is used whenever you want to equate a symbolic label with a numeric value, i.e., when you want to assign a particular value to a label so that you can refer to the value symbolically by the label, elsewhere in the program. An EQU statement is one way that an assembly language programmer can define a constant. The value of this pseudo-op is two-fold:

1) The use of labels should make the program easier to understand, both for the original programmer and anyone else looking at the source listing at some later time.

2) The program is much easier to modify, simply by changing the value in the EQU statement.

For example, let us assume that we initially set up a delay value which is perhaps used several times elsewhere in the program as a counter for a time delay routine. We could have the following statement in our source code:

```
00100   DELAY    EQU   128
```

After assembling our program and running it, we find that the original value of 128 is not optimum. We can then edit the EQUate statement to adjust the value in the desired direction. We then reassemble and re-test, etc.

The value given in an EQU statement can be a 16-bit number; a label can be equated to a particular address, for instance. Be aware, however, that the assembler will not allow you to specify loading a value greater than 8 bits into an 8-bit register.

Since the EQU statement defines a constant, it is logical that the program may contain only one EQU for any given label. It can be placed anywhere in the source program. I have formed the habit of placing EQU statements near the front of the program since it helps to have that information as you are perusing a source listing (especially someone's other than your own). That is strictly my personal preference; most assemblers will allow EQU statements to be placed anywhere in the source program. Choose your own approach and maintain that approach consistently from program to program.

DEFL Pseudo-Op

A label

The DEFL is somewhat similar to the EQU statement in that it is used to equate a symbolic label with a particular numeric value. It differs from the EQU in that you can have multiple DEFL statements for a given label.

In a sense, then, you can use the DEFL statement to define "temporary" constants. In my opinion this can be more trouble than it is worth, and I use the DEFL pseudo-op very seldomly when using an ordinary assembler. With a macro assembler which has conditional assembly and other features too advanced for us to consider now, good use can be made of DEFL. For you at this phase of your study, I recommend not attempting to use the DEFL in your programming efforts.

ΔFINIS & start

END Pseudo-Op

For most assemblers, there must be some way of saying that this is all you want assembled in this module. It normally is at the end of the source code (maybe that's why they call it END?) as the last statement. For ordinary assemblers the END statement has one further use, and that is to tell the assembler what the execution starting address is. For example, if you had the following code sequence:

```
00010          ORG   7F00H
00020  START   LD    A,5
                .
                .
                .
00200          END   START
```

The assembler would write the object code onto tape or disk in such a manner as to indicate that execution should begin at START, which in this example is 7F00H. If you specify no label after the END, the assembler sets up 0 as the starting address for the tape or disk object file accordingly.

Although not fatal, this is often inconvenient! If you have a tape system, you normally load a machine language program using the SYSTEM command in Level II BASIC. Then you give it a "/" to begin execution. If there was a valid starting address written to the tape originally, you need not specify a starting address. If there was none, execution will begin at 0, which causes re-initialization of the system, unless you give the valid address after the "/".

The problem is even worse when you want to load and execute an object program which resides on disk. If you just key in the program's name followed by <ENTER> while in DOS command mode (the customary procedure), the system will re-boot! You will have to do something involved like activating the Debug utility before typing in the program name as above. When the Debug display appears, you must specify the desired "Go" address to begin execution. It is much simpler to specify the label after the END! 'Nuff said!

DEFB Pseudo-Op

This is used to define a byte of data which is actually to be placed in the object file at the current address. This differs from the DEFL and the EQU pseudo-ops, which just tell the assembler to substitute the numeric value whenever the label is encountered. The DEFB statement causes the value specified to be deposited in the object file immediately, and is therefore useful for initializing a byte of data which perhaps later will change value, etc. You will see examples of use of the DEFB pseudo-op in various routines we will be examining.

DEFW Pseudo-Op

DEFW is similar to DEFB except that it defines a word of 16-bits, i.e., two bytes of data. In accordance with the usual procedure for a Z80 processor, these are placed in memory with the least significant byte first. For example, if you had the following statement in your program:

```
DEFW  4000H
```

the 00 would be stored first and the 40H would be stored at the next higher memory address.

DEFM Pseudo-Op

This is used to define a "message" or string of ASCII characters enclosed in single quotes. The numeric values of the characters are placed one after the other in sequential order in memory.

DEFS Pseudo-Op

This defines a storage area of nn bytes, where "nn" is a number following the DEFS. The DEFS simply causes the assembler to add "nn" to the current address, leaving a gap which can be used during execution of the program to store data. etc. It is important to realize that most assemblers do not initialize the memory in this area to any particular values, so whatever garbage is in the area will be in your object code when you begin execution. The exact handling of DEFS differs from one assembler to another, but I can almost guarantee that "garbage" will be present when execution begins. Of course, this is of no consequence if you use the area of memory properly during program execution, that is, if you store valid data there before any routine retrieves any data. Otherwise, the ancient saying "Garbage In, Garbage Out" becomes personally meaningful!

Miscellaneous Loose Ends

Assemblers differ in statement syntax regarding labels. Many assemblers require that any statement defining a label have the first character of the label in the first column position of the line; there can be no space characters or tabs, etc., before the label.

Some assemblers require that you terminate the label with a colon. The Radio Shack macro assembler is like this. Normally when the terminator is required, you gain something in return, like not having to place the first character of the label in column one; i.e., the statement can be free-format. Again, simply check the manual.

Most assemblers allow you to reference the current value of the pseudo program counter in some manner. The most common way this is accomplished is by using a statement like:

```
MYLABL   EQU   $
```

Check the documentation to see how your assembler implements this. An example of its use with the Series I Editor/Assembler is line 1180 of the QDMTST program in Chapter 3 of this manual.

Most assemblers have various other statements useful during assembly which I have not covered above. Let's talk about them now, in no particular order of importance.

For example, the Series I assembler has "*LIST ON" and "*LIST OFF" commands which allow you to inhibit the output of any portion of the source listing you wish. You may want to do this if you have quite a few messages or other data in your program -- after you have once checked them to be sure they are correct, you may wish to inhibit their appearance in later listings, to save both time and paper.

Expressions can be used in place of a numeric operand with most assemblers. Any label used in the expression must be defined somewhere in the program. Almost all assemblers allow use of the "+" and "-" signs for their conventional purposes -- designation of addition, subtraction, or negation.

Some assemblers provide for limited Boolean arithmetic to calculate expressions. The Series I allows only the logical AND, designated by the "&" character.

A very few assemblers allow multiplication or division in expression evaluations. Some assemblers do allow you to specify arithmetic shifting, which can effect integer multiplication or division by powers of 2. The Series I allows you to designate shifting with the "<" character. If the value following the "<" is positive, a one-bit left shift will be performed for the number of times given. If the value is negative, a series of one-bit right shifts will be performed. For example, given that:

```
VALUE1   EQU   4
VALUE2   EQU   VALUE1<2
VALUE3   EQU   VALUE1<-1
```

binary dec
100 = 4
10000 = 16
10 = 2

After the evaluations of the expressions in the second and third lines above, the assembler will have equated VALUE1 with 4, VALUE2 with 16, and VALUE3 with 2. Try it, you'll like it!

Most assemblers allow for character constants. Suppose you wanted to load the accumulator with an ASCII "A". You could look in a table to find that "A" in the ASCII character set has a numeric value of 41H and use that value as in this:

```
LD    A,41H
```

The Series I allows you to use this instead:

```
LD   A,'A'
```

where the single quotes tell the assembler to convert that character into its ASCII value.

Almost all assemblers allow use of different number bases, or radices, when specifying numbers. Usually the default is radix 10 (decimal). If a particular number does not have one of the following suffixes, the Series I assembler will interpret it as a decimal number:

"H" suffix designates radix 16 (hexadecimal). You will use hex numbers frequently.

"Q" suffix designates radix 8 (octal). Some assemblers also permit the use of the letter "O" for octal, but I do not recommend its use because it is too easily mistaken for the number zero by merely mortal humans.

"B" suffix. A few assemblers (not the Series I) allow for radix 2 (binary).

All assemblers allow comments in assembly statements; the usual character to designate a comment is the semi-colon, but sometimes the asterisk is used for this purpose. The assembler will not try to interpret any characters after the semi-colon as an assembly language instruction. I highly recommend heavy use of comments in your programs, for obvious reasons.

PROGRAMMING EXAMPLE

We now have an overview of an assembler. Let us now look at a trivial program, in several stages of development. The program gives us the title of this chapter; it is designed to display "The Cat Chased The Mouse!" on the screen. Its sole purpose is to help you get familiar with your particular editor/assembler. You should key in the program and assemble it, making any and all changes necessary to get it to assemble without error. As I have threatened several times, this may require you to read the manuals which came with your editor/assembler! Never fear -- plunge right in. You will find me very patient -- take as long as you need, since you MUST learn to use your assembler before we can make significant progress from here on.

Since we are going to send a message to the screen, let us get a little background information about screen memory. The TRS-80 screen memory is ordinary memory, in that it can be written to or loaded from in the same way as any other RAM. Screen memory is located between 3C00-3FFFH. As you become more at ease using assembly language with your system, you may want to experiment with loading directly to and from screen memory, but for the time being let's use some ROM routines and Level II storage locations to keep from getting bogged down with details, details, details. You will learn much more about memory mapping and screen memory in Chapter 5.

15360-16383

Here's our first draft of the program:

```
00010                ORG   5200H          ;Program load addr
00020     TV         EQU   33H            ;ROM Display Pgrm.
00030     START      LD    HL,MSG         ;Pnt to "Cat", etc.
00040     MSGOUT     LD    A,(HL)         ;Fetch char in msg
00050                AND   A              ;Chk for zero value
00060                JR    Z,ZEND         ;if so, finished
00070                CALL  TV             ;Display byte in A
00080                INC   HL             ;Pnt to nxt char
00090                JR    MSGOUT         ;Try it again
00100     ZEND       JR    ZEND
00110     ;
00120     MSG        DEFM  'The Cat Chased the Mouse!'
00130                DEFB  0
00140                END   START
```

Well, that doesn't appear too hard, does it? Let's look at what I have done. The first thing I did is to tell the assembler (I am using the Series I assembler mentioned earlier) that I want the object code to be assembled to execute starting at 5200H, the lowest address I can use with a disk system. Once again, if you have a Model III disk system, you may want to use 5600H. Use 4400H for a tape-based system. I then define the label "TV" in line 20. Starting in line 30 is the actual routine which will output the message to the screen. I have labelled the statement "START" since this is the start of the routine; sometimes my imagination in choosing label names is overwhelming! I use HL reg pair to point to the message. WHY, you ask? Well, I am pointing to an address, and that requires a 16-bit value to describe a unique address. WHY THE HL REG PAIR, you ask? Well, partly out of habit, I must admit! However, the habit was consciously cultivated since the Z80 instruction set favors the HL reg pair regarding both execution time and memory utilization. In this case, I could just as easily have used the BC or the DE reg pairs. I'll show you why I can say that.

Look at the statement labelled MSGOUT (line 40). I need to send the characters of the message to the ROM routine ("TV", which is located at 33H). For any Z80 machine, almost all I/O must pass through the accumulator; this is a result of the way the instruction set is structured. The TV routine will output whatever character is in the accumulator to the screen. So we know we have to fetch the character from memory into the A reg.

Look on page B-2 of this manual. The third instruction listed on that page is "LD r,(HL)". This is the instruction we used, with A the reg used in place of r. This instruction uses 1 byte of memory and takes 7 T-cycles to execute. We could have used either of the next two instructions shown, "LD r,(IX+d)" or "LD r,(IY+d)", but they each take 3 bytes and 19 T-cycles! So we waste both memory and time with the index regs. Don't use the index regs for this type of problem unless you absolutely don't have any reg pairs available!

On down the page (about half way), you will see "LD A,(BC)" and "LD A,(DE)". These take 1 byte and 7 T-cycles, just like the instruction using HL. So anytime you want to bring a character into the A reg, consider either of these three instructions as equivalent. Aside. . . the instruction using the HL reg pair is more versatile than the ones for the BC and DE reg pairs, since you can put the byte into ANY of the 8-bit regs if you use the HL reg pair as memory pointer, but into only the accumulator if you use either the BC or DE reg pairs.

So far we have gotten the first char into the A reg. Now we must check it to see if it is our "end of message" byte, which is necessary to exit from the routine. I have set up this delimiter as 0, but it could be any character not appearing in the message itself. WHY DID I CHOOSE A ZERO, you ask? Because there are a couple of neat, 1-byte instructions to allow me to test for a zero value in the A reg. On page B-5, about half way down, you will find "AND s" and "OR s"; these Boolean operations can be used to exercise the flags if I substitute "A" for "s". The net result is an unchanged A reg with the Zero flag reflecting whether or not the value in the A reg is 0. If it is 0, we are finished and jump relative to ZEND.

If it is not 0, we are ready to call the TV routine to display the character. We then increment our pointer to the message (HL reg pair) and loop back ad infinitum if there is no zero byte at the end of our message!

When we finally get our zero byte, we execute the instruction at ZEND, which says to jump relative to ZEND! How absurd! Doesn't the computer have anything better to do? No, not in this case. Remember, the CPU must ALWAYS be executing SOME instruction all of the time the system is powered up. Even the HALT instruction is executed as an endless string of NOP instructions; only an interrupt will cause execution of the instruction following the HALT.

The human reading the message is not fast enough to read it if the computer immediately does something like erase the screen or list a program, etc., without waiting. The advantage of the instruction in line 130 above is that even an interrupt will not break out of this loop. This will stay there until you press the Reset button. Read at your leisure, you might say!

Key the program in, assemble it, and run it. I'll wait right here till you are finished.

It's not very elegant, is it? It just prints the message wherever it left off printing whatever came before! And who wants to press the Reset button to get out of a program?

Let's fancy it up a bit. Why don't we clear the screen first? O.K., all we have to do is add one instruction and one EQU. In making additions to the previous version of the program, I will maintain the same line numbers for a given original instruction; the new instructions will have line numbers between those of the original instructions. This is done simply to make the example easier to follow.

Ordinarily you would renumber the program periodically. Here's what the program looks like now:

```
00010              ORG   5200H
00020     TV       EQU   33H
00022     CLS      EQU   1C9H
00030     START    LD    HL,MSG          ;Pnt to "Cat", etc.
00032              CALL  CLS             ;Clear the screen
00040     MSGOUT   LD    A,(HL)          ;Fetch char in msg
00050              AND   A               ;Chk for zero value
00060              JR    Z,ZEND          ;if so, finished
00070              CALL  TV
00080              INC   HL              ;Pnt to nxt char
00090              JR    MSGOUT          ;Try it again
00100     ZEND     JR    ZEND
00110     ;
00120     MSG      DEFM  'The Cat Chased the Mouse!'
00130              DEFB  0
00140              END   START
```

Try that -- I'll wait.

Although that's certainly better, maybe we want to print the message in the center of the screen. Hey, that's a good idea! The first thing we must do is figure out where the center of the screen is. Well, let's not do exactly that -- let's be even fancier and center the message on the line near the center of the screen. (I say "near the center" of the screen because there are 16 lines of 64 characters each, totalling 1024 chars. No line is at the exact center of the screen! But close enough for friends like us, right?) Counting the characters in the message, I get 25 -- don't count the 0, since it won't be displayed. So if we go down to the 8th line, that will be 7 * 64 = 448 to the start of the line (line numbering starts with 0). Now 64 - 25 = 39. If we split the difference, we need either 19 or 20 spaces before our message (to exactly center it we need 19.5, but who can do that?) So let's use 19 -- our total offset from the top left of the screen is then 448 + 19 = 467. Let's let the assembler do some of our work for us -- we will give it the start of memory as an EQU and tell it to add 467 to that. Then we will tell it to update the cursor position before displaying the message. This uses a system RAM location at 4020H, which I have labelled "CURSOR" in an EQU. You may find this address very useful to you in your own programming, since it allows you to perform a PRINT@ function in assembly language, as this program illustrates.

Here's our program so far:

```
00010                ORG   5200H
00020      TV        EQU   33H
00022      CLS       EQU   1C9H
00024      CURSOR    EQU   4020H
00026      TVSTRT    EQU   3C00H
00030      START     LD    HL,MSG          ;Pnt to "Cat", etc.
00032                CALL  CLS             ;Clear the screen
00034                LD    DE,TVSTRT+467   ;center the msg
00036                LD    (CURSOR),DE
00040      MSGOUT    LD    A,(HL)          ;Fetch char in msg
00050                AND   A               ;Chk for zero value
00060                JR    Z,ZEND          ;if so, finished
00070                CALL  TV
00080                INC   HL              ;Pnt to nxt char
00090                JR    MSGOUT          ;Try it again
00100      ZEND      JR    ZEND
00110      ;
00120      MSG       DEFM  'The Cat Chased the Mouse!'
00130                DEFB  0
00140                END   START
```

Try that now -- I don't mind waiting one more time.

This is getting fancy! But we still need to arrange a better way of ending the routine so that we do not have to press the Reset button every time. A convenient way to do that is to check the keyboard to see if a key is depressed -- if a key is not depressed, make it stay in the loop; when a key is pressed, we'll jump back to the system (either Level II BASIC or the disk operating system as appropriate) for our next bit of excitement. On the next page is the final version for this session:

```
00010                ORG    5200H
00020      TV        EQU    33H
00022      CLS       EQU    1C9H
00024      CURSOR    EQU    4020H
00026      TVSTRT    EQU    3C00H
00027      KEYIN     EQU    2BH
00028      SYSTEM    EQU    402DH          ;Entry pnt for DOS
00029                                       ;1A19H for Level II
00030      START     LD     HL,MSG         ;Pnt to "Cat", etc.
00032                CALL   CLS            ;Clear the screen
00034                LD     DE,TVSTRT+467  ;center the msg
00036                LD     (CURSOR),DE
00040      MSGOUT    LD     A,(HL)         ;Fetch char in msg
00050                AND    A              ;Chk for zero value
00060                JR     Z,ZEND         ;if so, finished
00070                CALL   TV
00080                INC    HL             ;Pnt to nxt char
00090                JR     MSGOUT         ;Try it again
00100      ZEND      CALL   KEYIN
00102                OR     A
00104                JR     Z,ZEND
00106                JP     SYSTEM
00110                ;
00120      MSG       DEFM   'The Cat Chased the Mouse!'
00130                DEFB   0
00140                END    START
```

Well, I think you have done very well! Now that you know how to use your assembler, we can progress to more involved routines. The program listings begin in Chapter 3, with the discussion beginning in Lesson 6 of the audio lectures. I'm ready whenever you are, but don't be in too much of a rush. It is best for you to understand things as you go, so don't exceed your optimum speed.

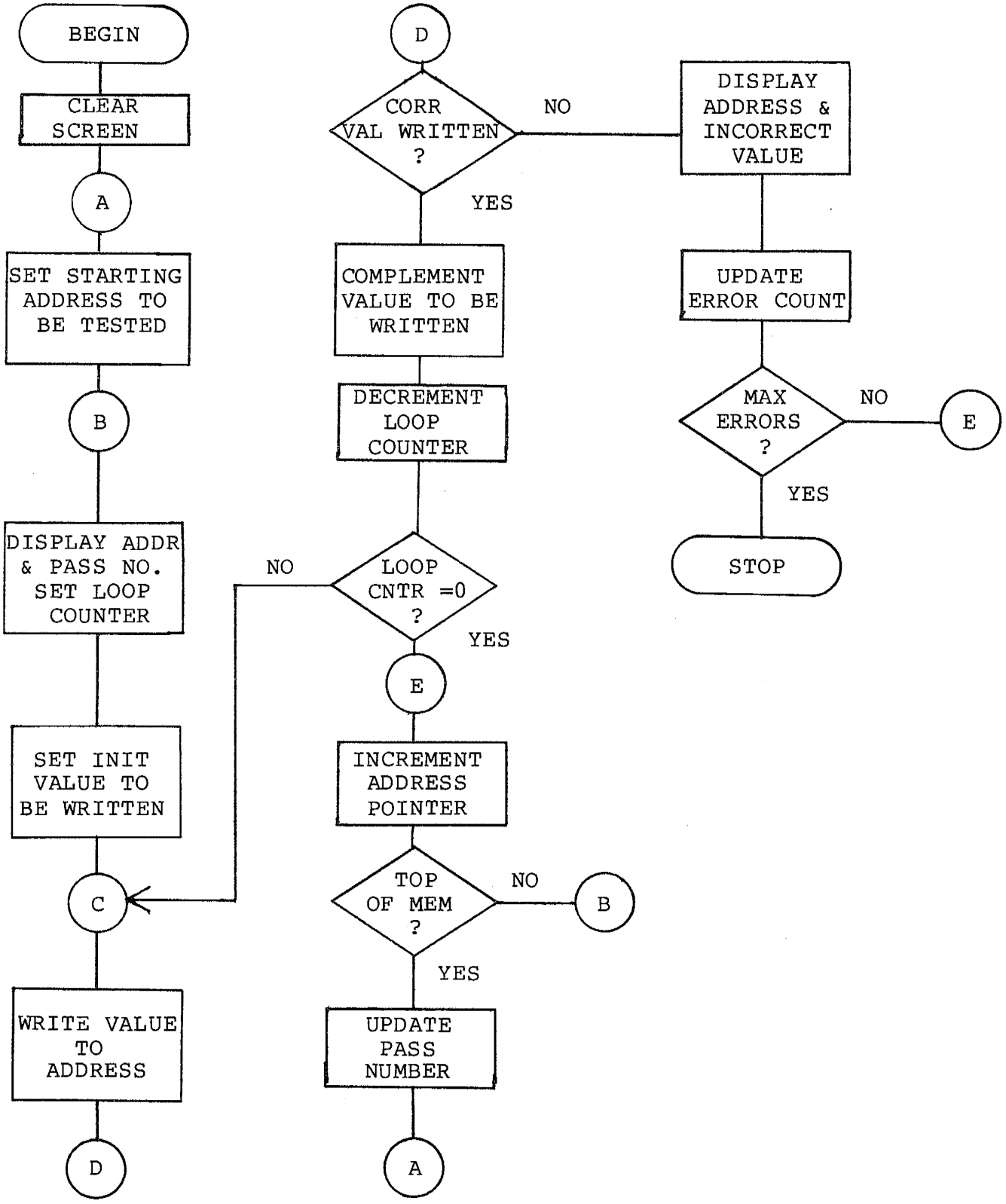
CHAPTER 3
QUICK AND DIRTY MEMORY TEST ROUTINE

The "quick and dirty" memory routine is the first program to be discussed step-by-step in the audio lessons (lessons 6 and 7). The purpose of the routine is to test RAM memory for errors, and illustrates how to string assembly language instructions together to form a program.

The test itself is relatively simple -- first a byte of zeroes is written to a memory location and then read to ascertain that the correct value is obtained. Then a byte of ones is written and checked. This sequence continues for 127 more times for that memory location and then the next location is checked in like manner. The program displays the address of the memory under test and the number of passes which has bee made through the entire memory. If an error is found, both the address of that error and the byte of data obtained from that address are displayed.

When all of the available RAM memory has been tested, the program begins the next pass. The only ways to stop the execution of the program are to press the <RESET> button or to have enough memory errors to fill the screen.

FLOW CHART


BEGIN
CLEAR SCREEN
A
SET STARTING ADDRESS TO BE TESTED
B
DISPLAY ADDR & PASS NO. SET LOOP COUNTER
SET INIT VALUE TO BE WRITTEN
C
WRITE VALUE TO ADDRESS
D
D
CORR VAL WRITTEN ?
NO
YES
DISPLAY ADDRESS & INCORRECT VALUE
COMPLEMENT VALUE TO BE WRITTEN
UPDATE ERROR COUNT
DECREMENT LOOP COUNTER
MAX ERRORS ?
NO
E
YES
STOP
LOOP CNTR =0 ?
NO
YES
E
INCREMENT ADDRESS POINTER
TOP OF MEM ?
NO
B
YES
UPDATE PASS NUMBER
A

```
                    00010 ;
                    00020 ; 'QDMTST' IS A "QUICK AND DIRTY" MEMORY TEST ROUTINE.
                    00030 ; IT TESTS ALL MEMORY FROM JUST PAST QDMTST TO THE
                    00040 ; TOP OF MEMORY.
                    00050 ;
                    00060 ; THE TEST IS QUITE SIMPLE: FIRST A BYTE OF ZEROES
                    00070 ; IS WRITTEN TO THE LOCATION AND CHECKED; THEN A
                    00080 ; BYTE OF ONES (0FFH) IS WRITTEN AND CHECKED.
                    00090 ; THIS SEQUENCE IS REPEATED 127 TIMES BEFORE
                    00100 ; GOING ON TO THE NEXT LOCATION.
                    00110 ;
                    00120 ; AFTER COMPLETING A PASS, THE PROGRAM STARTS THE
                    00130 ; NEXT ONE.  THE ONLY WAYS TO STOP THE PROGRAM ARE
                    00140 ; EITHER TO HIT THE RESET BUTTON OR HAVE ENOUGH
                    00150 ; MEMORY ERRORS TO FILL THE SCREEN (HEAVEN FORBID!).
                    00160 ;
0033                00170 CRT      EQU     33H
0077                00180 MXERRS   EQU     119
4020                00190 TVCURS   EQU     4020H
40B1                00200 TOPMEM   EQU     40B1H
001C                00210 HOME     EQU     1CH
001F                00220 EREOF    EQU     1FH
                    00230 ;
5200                00240          ORG     5200H           ;42E9H LOWEST FOR TAPE
                    00250 ;                                ;5600H FOR MOD 3 DISK
5200 3E1C           00260 QDMTST:  LD      A,HOME          ;CLEAR SCREEN
5202 CD4F52         00270          CALL    TV
5205 3E1F           00280          LD      A,EREOF
5207 CD4F52         00290          CALL    TV
520A DD218D52       00300          LD      IX,PASS         ;PNT TO PASS NO.
520E 219152         00310 TEST:    LD      HL,START        ;1ST ADDR TO BE TESTED
5211 CD5752         00320 LOOP1:   CALL    DSPLAY          ;DISPLAY ADDR
5214 AF             00330          XOR     A               ;INIT CHAR TO 0
5215 47             00340          LD      B,A             ;SET LOOP CNTR TO 256
5216 77             00350 LOOP2:   LD      (HL),A          ;WRITE CHAR TO MEM
5217 BE             00360          CP      (HL)            ;WRITE OK?
5218 2012           00370          JR      NZ,ERROR        ;NO
521A 2F             00380          CPL                     ;YES,FORM OPPOSITE
521B 10F9           00390          DJNZ    LOOP2           ;PATTERN & CONTINUE
521D 23             00400 ENDTST:  INC     HL              ;NXT ADDR TO BE TESTED
521E EB             00410          EX      DE,HL           ;IS IT PAST THE TOP OF
521F 2AB140         00420          LD      HL,(TOPMEM)     ;UNPROTECTED MEMORY?
5222 ED52           00440          SBC     HL,DE
5224 EB             00450          EX      DE,HL
5225 20EA           00460          JR      NZ,LOOP1        ;NO,TEST THIS ADDR
5227 DD3400         00470          INC     (IX+0)          ;YES,INC PASS NO.
522A 18E2           00480          JR      TEST            ;& HAVE ANOTHER GO!
522C E5             00490 ERROR:   PUSH    HL
522D 2A8E52         00500          LD      HL,(TVPNTR)     ;SET TV CURSOR
5230 110800         00510          LD      DE,8            ;TO NXT 8-BYTE
5233 19             00520          ADD     HL,DE           ;AREA AVAILABLE
5234 222040         00530          LD      (TVCURS),HL
5237 228E52         00540          LD      (TVPNTR),HL     ;SAVE NEW POINTER
```

```
523A E1       00550          POP     HL
523B CD6652   00560          CALL    DSPADR          ;DISPLAY ADDR OF ERROR
523E 7E       00570          LD      A,(HL)          ;GET FAULTY BYTE
523F CD7452   00580          CALL    HXCHAR          ;DISPLAY IT
5242 3A9052   00590          LD      A,(ERRORS)      ;UPDATE ERROR CNT
5245 3C       00600          INC     A               ;UPDATE ERROR COUNT
5246 329052   00610          LD      (ERRORS),A
5249 FE77     00620          CP      MXERRS          ;ALL SCREEN CAN HOLD?
524B 20D0     00630          JR      NZ,ENDTST       ;NO, PROCEED
524D 18FE     00640 STOP:    JR      STOP            ;YES: WOW, DID I MESS UP!
              00650 ;
              00660 ; OUTPUTS CHAR IN ACC TO THE TV SCREEN
              00670 ;
524F F5       00680 TV:      PUSH    AF
5250 D5       00690          PUSH    DE
5251 CD3300   00700          CALL    CRT
5254 D1       00710          POP     DE
5255 F1       00720          POP     AF
5256 C9       00730          RET
              00740 ;
              00750 ; DISPLAYS CURRENT MEMORY ADDRESS AND PASS COUNT
              00760 ; IN TOP LEFT PORTION OF SCREEN
              00770 ;
5257 3E1C     00780 DSPLAY:  LD      A,HOME          ;RESET CURSOR TO
5259 CD4F52   00790          CALL    TV              ;TOP LEFT OF SCREEN
525C CD6652   00800          CALL    DSPADR          ;DISPLAY CURRENT MEM ADDR
525F DD7E00   00810          LD      A,(IX+0)        ;DISPLAY PASS COUNT
5262 CD7452   00820          CALL    HXCHAR
5265 C9       00830          RET
              00840 ;
              00850 ; DISPLAY MEMORY ADDRESS OF CURRENT LOCATION
              00860 ;
5266 7C       00870 DSPADR:  LD      A,H             ;DISPLAY MS BYTE
5267 CD7452   00880          CALL    HXCHAR
526A 7D       00890          LD      A,L             ;LS BYTE
526B CD7452   00900          CALL    HXCHAR
526E 3E20     00910          LD      A,' '           ;OUTPUT A SPACE
5270 CD4F52   00920          CALL    TV
5273 C9       00930          RET
              00940 ;
              00950 ; DISPLAYS BYTE IN ACC AS TWO HEX CHARS ON TV
              00960 ;
5274 C5       00970 HXCHAR:  PUSH    BC
5275 4F       00980          LD      C,A             ;TEMP SAVE CHAR
5276 0F       00990          RRCA                    ;MOVE RIGHT 4 BITS
5277 0F       01000          RRCA                    ;TO GET MOST
5278 0F       01010          RRCA                    ;SIG HEX CHAR
5279 0F       01020          RRCA                    ;IN PLACE
527A CD7F52   01030          CALL    HXTV            ;OUTPUT IT
527D 79       01040          LD      A,C             ;RETRIEVE CHAR
527E C1       01050          POP     BC
527F E60F     01060 HXTV:    AND     0FH             ;STRIP 4 MSB'S
5281 FE0A     01070          CP      10              ;DIGIT?
```

```
5283 3802      01080          JR      C,HT0           ;YES
5285 C607      01090          ADD     A,7             ;NO,ADD OFFSET FOR ALPHA
5287 C630      01100 HT0:     ADD     A,30H           ;CNVRT TO ASCII
5289 CD4F52    01110          CALL    TV              ;OUTPUT IT
528C C9        01120          RET
               01130 ;
528D 01        01140 PASS:    DEFB    1               ;UPDATED FOR EACH PASS
528E 003C      01150 TVPNTR:  DEFW    3C00H           ;UPDATED FOR EACH ERROR
5290 00        01160 ERRORS:  DEFB    0               ;UPDATED FOR EACH ERROR
               01170 ;
5291           01180 START    EQU     $               ;TESTING STARTS HERE
               01190 ;
5200           01200          END     QDMTST

00000 Total Errors
```

CHAPTER 4
KBDTVMOD ROUTINE

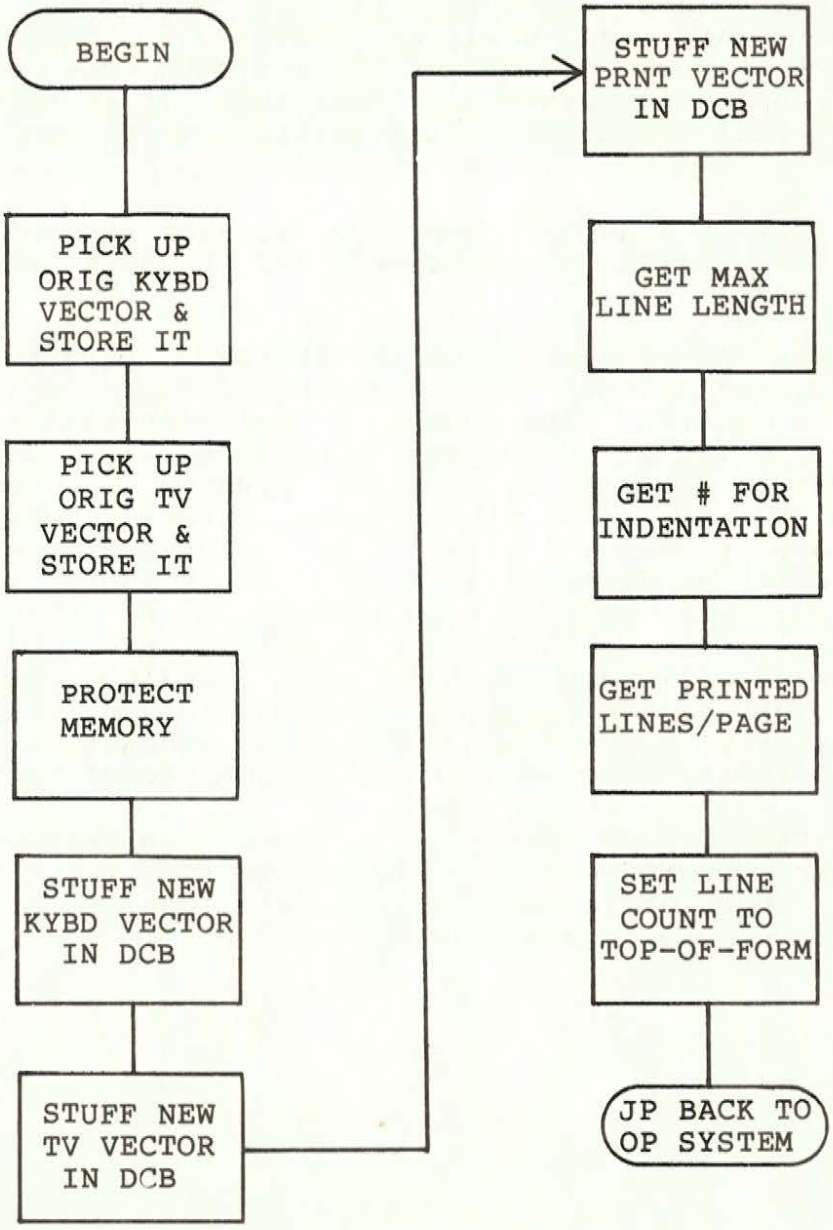
This routine illustrates how the input/output drivers can be modified to perform special tasks the programmer desires. The particular example provides a routine which intercepts characters being sent to the screen and sends them to the printer as well, upon command. Although this function may not now be needed if you have one of the latest disk operating systems, I feel that it is excellent from the tutorial standpoint and is well worth our time discussing it.

Audio lessons 8 and 9 cover the routine step-by-step. This chapter contains the documentation which is referred to in my discussion.

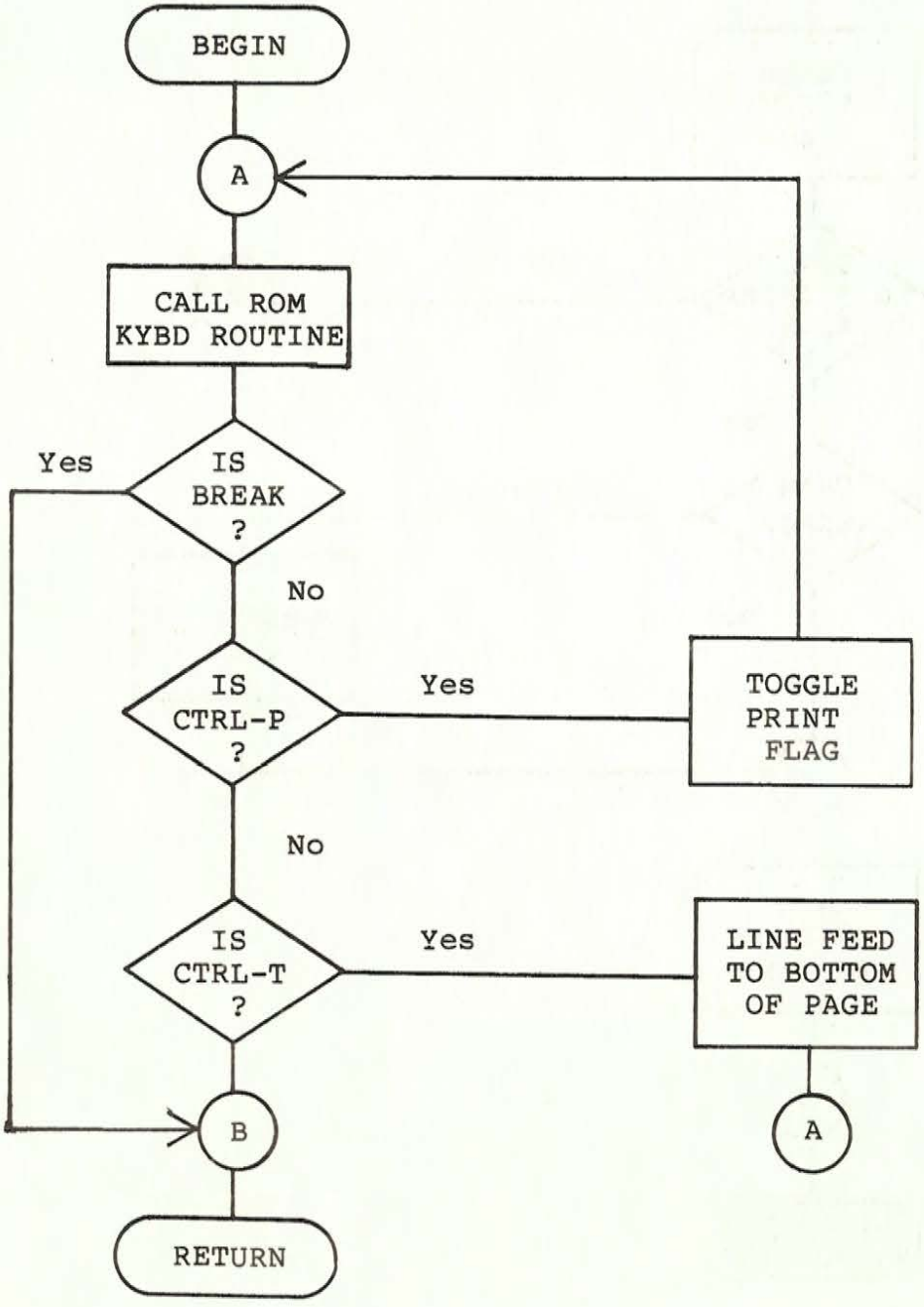
Here is a brief summary of the KBDTVMOD program. The keyboard, TV, and printer device control blocks are modified to accomplish the task. By depressing the <DOWNARROW>, <SHIFT>, and <P> keys simultaneously, the user can alternately enable or disable sending a character destined to the screen to the printer as well. When the three keys are pressed the first time, all characters to be displayed will be directed also to the printer; pressing them once more will disable this function. In the current implementation, any special character less than an ASCII space character will be converted to a carriage return before being sent to the printer. This avoids possible misbehavior by your printer upon receiving characters which are special display functions -- CLS, EREOL, EREOF, etc.

The printer can be sent to the top of the next page by depressing the <DOWNARROW>, <SHIFT>, and <T> keys simultaneously. There are some enhancements to this routine suggested at the end of audio lesson 9.

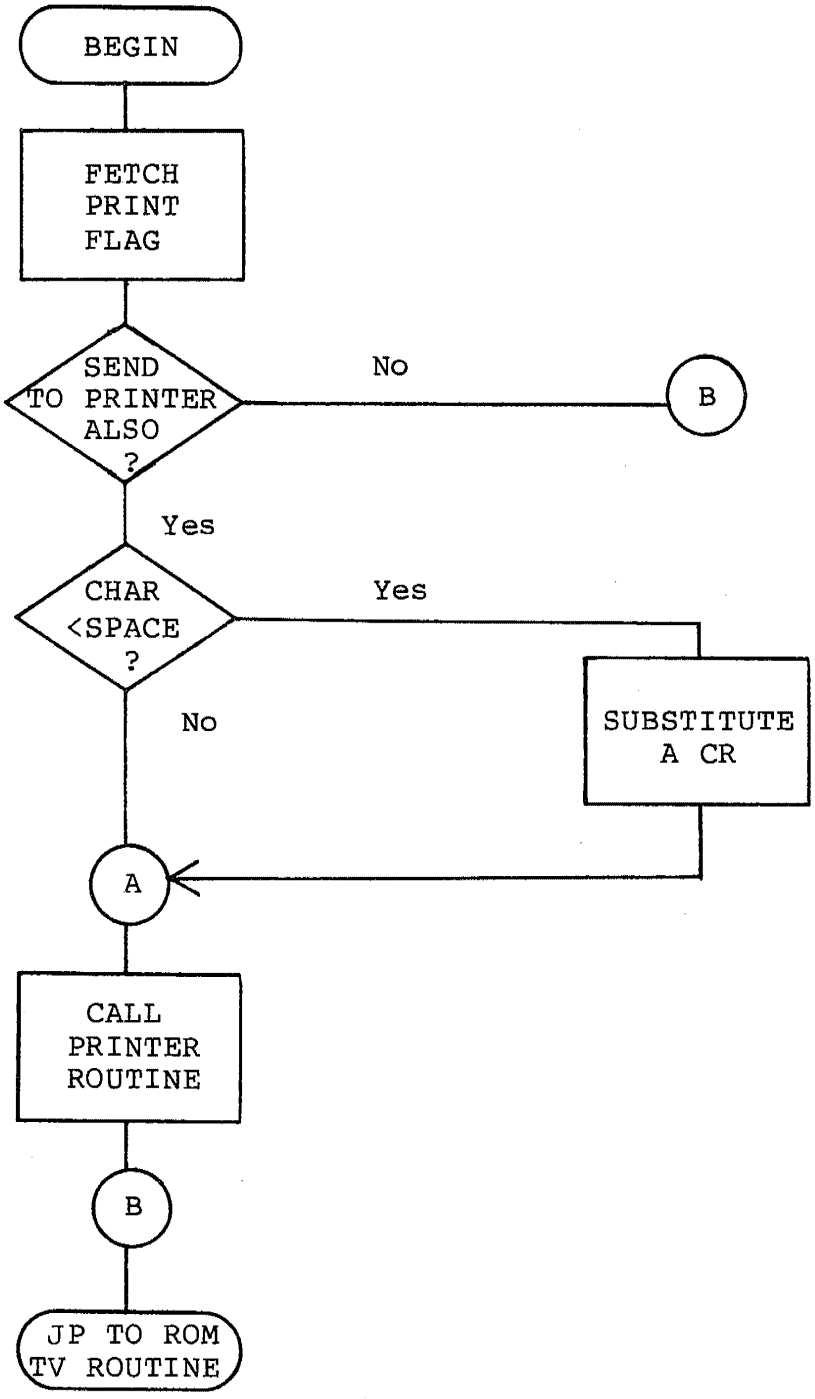
FLOW CHART 1 OF 6
Initialization Segment



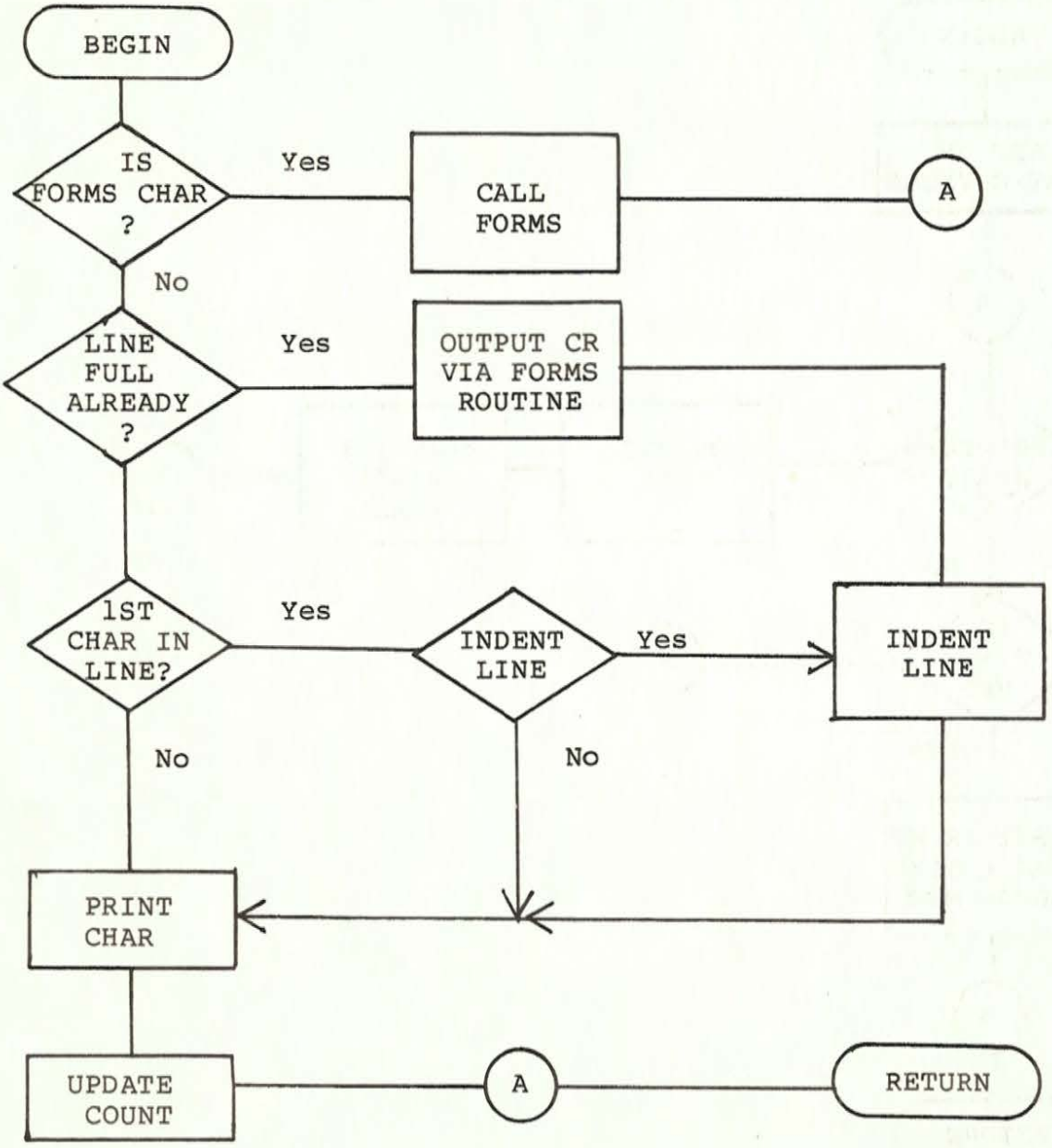
FLOW CHART 2 OF 6
KBDMOD Segment



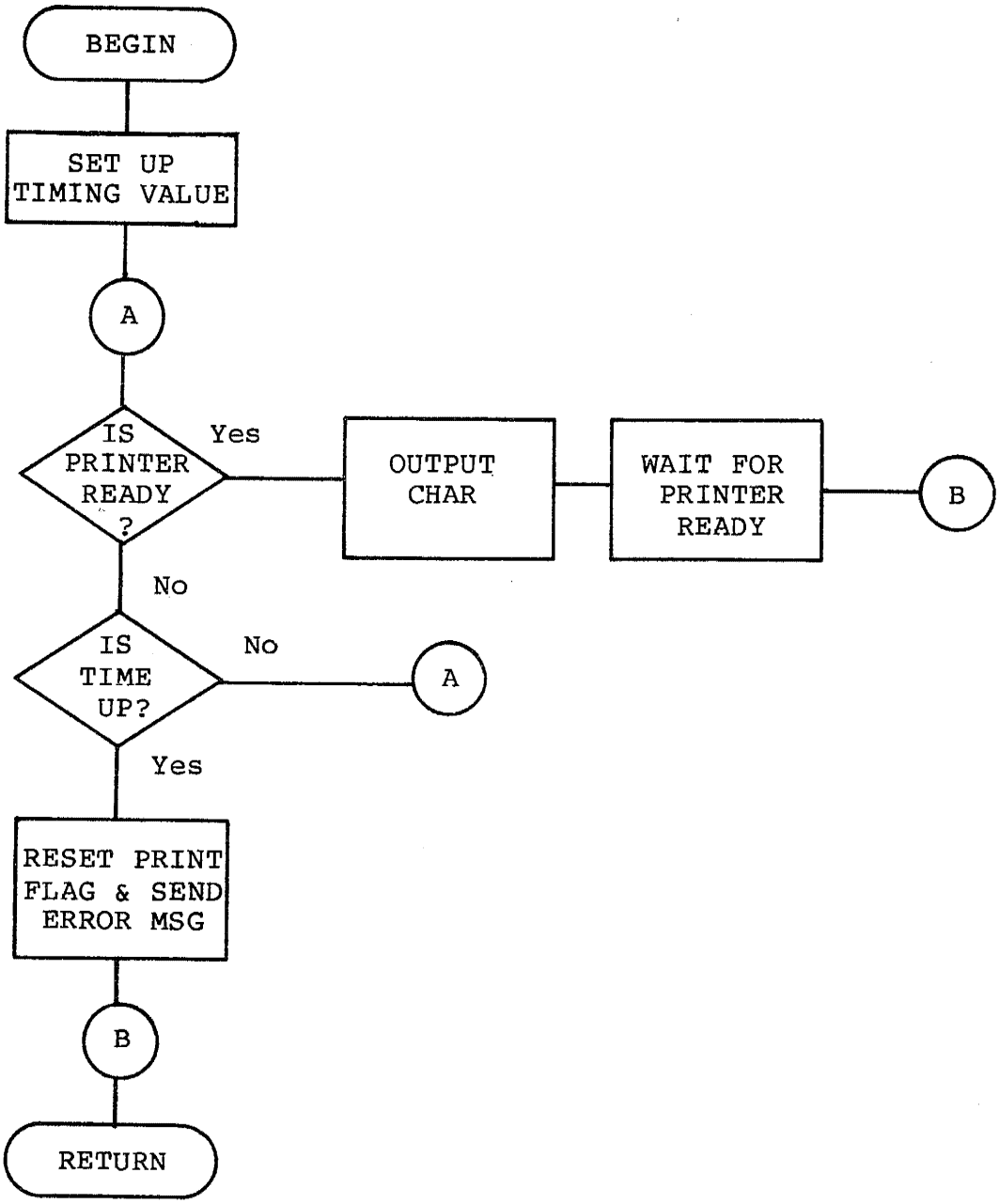
FLOW CHART 3 OF 6
TVMOD Segment



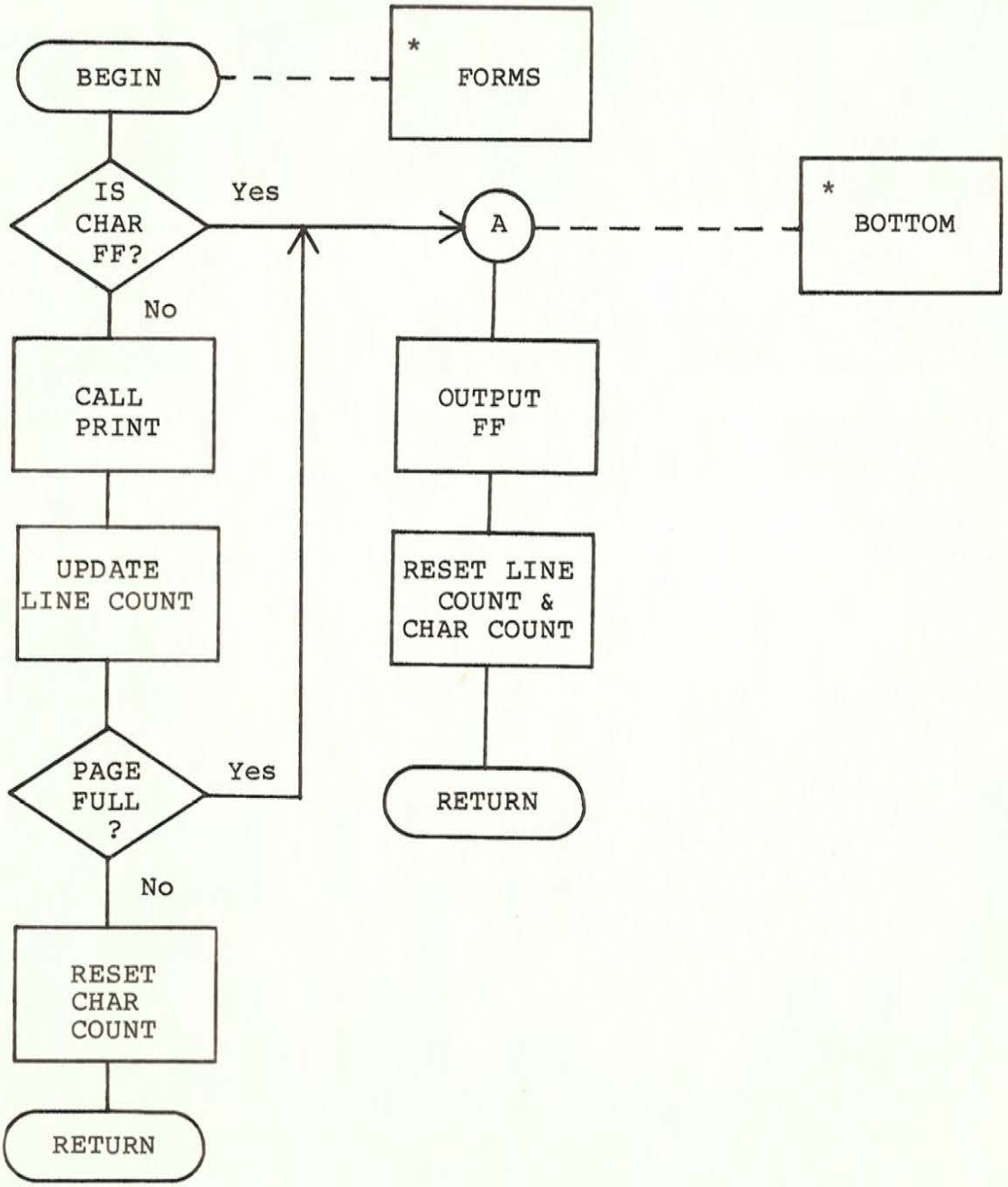
FLOW CHART 4 OF 6
Print Handler



FLOW CHART 5 OF 6
PRINT Routine



FLOW CHART 6 OF 6
FORMS & BOTTOM Routines



NOTES

```
                00010 ;
                00020 ; 'KBDTVMOD' MODIFIES BOTH THE KEYBOARD AND TV ROUTINES
                00030 ; SO THAT BY DEPRESSING THE DOWNARROW, SHIFT, AND "P" KEYS
                00040 ; SIMULTANEOUSLY THE USER CAN ALTERNATELY ENABLE OR
                00050 ; DISABLE SENDING A CHARACTER DESTINED FOR THE TV TO THE
                00060 ; PRINTER AS WELL.  IN THE CURRENT IMPLEMENTATION, ANY
                00070 ; NON-PRINTING CHARACTER LESS THAN A SPACE IS CONVERTED
                00080 ; TO A CR-LF COMBINATION.  THIS AVOIDS COMPLEXITIES
                00090 ; INCURRED WITH THE CLS FUNCTION, EREOL, OR EREOF, ETC.
                00100 ; THE PRINTER CAN BE GIVEN A TOP-OF-FORM COMMAND FROM
                00110 ; THE KEYBOARD BY DEPRESSING THE DOWNARROW, SHIFT, AND
                00120 ; "T" KEYS SIMULTANEOUSLY.
                00130 ;
                00140 ; EQUATES
                00150 ;
4016            00160 KEYDRV  EQU     4016H
401E            00170 TVDRV   EQU     401EH
4026            00180 PRTDRV  EQU     4026H
402D            00190 SYSTEM  EQU     402DH           ;1A19H FOR LEVEL II
4049            00200 TOPMEM  EQU     4049H           ;MODEL I ONLY
40B1            00210 MEMTOP  EQU     40B1H
40A0            00220 STRING  EQU     40A0H
37E8            00230 PTSTAT  EQU     37E8H           ;PRINTER STATUS ADDR
37E8            00240 PRINTR  EQU     37E8H           ;FOR MODEL I ONLY
                00250 ;PRINTR EQU     0F8H            ;FOR MODEL III ONLY
0040            00260 LINEIN  EQU     40H
0033            00270 TV      EQU     33H
1E4F            00280 DECBIN  EQU     1E4FH
0010            00290 CTRLP   EQU     16
0014            00300 CTRLT   EQU     20
000D            00310 CR      EQU     0DH
000A            00320 LF      EQU     0AH
000C            00330 FF      EQU     0CH
0001            00340 BREAK   EQU     1
4029            00350 LINCNT  EQU     4029H
                00360 ;
FE00            00370         ORG     0FE00H
                00380 ;
FE00 2A1640     00390 START:  LD      HL,(KEYDRV)     ;PICK UP ORIG KYBD
FE03 2251FE     00400         LD      (KEYSTR),HL     ;VECTOR AND PLACE IT
FE06 2A1E40     00410         LD      HL,(TVDRV)      ;SAME FOR TV VECTOR
FE09 228BFE     00420         LD      (TVSTR),HL
FE0C 214FFE     00430         LD      HL,KBDMOD-1     ;PROTECT MEMORY
FE0F 224940     00440         LD      (TOPMEM),HL     ;MODEL I ONLY
FE12 22B140     00450         LD      (MEMTOP),HL
FE15 11CEFF     00460         LD      DE,0FFCEH       ;ALLOW FOR STRING MEM
FE18 19         00470         ADD     HL,DE
FE19 22A040     00480         LD      (STRING),HL
FE1C 2150FE     00490         LD      HL,KBDMOD
FE1F 221640     00500         LD      (KEYDRV),HL     ;SET NEW KYBD VECTOR
FE22 2170FE     00510         LD      HL,TVMOD        ;SET NEW TV VECTOR
FE25 221E40     00520         LD      (TVDRV),HL
FE28 218DFE     00530         LD      HL,PRNTR
```

```
FE2B 222640   00540          LD      (PRTDRV),HL
FE2E 2152FF   00550          LD      HL,LENMSG        ;ASK FOR MAX LINE LEN
FE31 CD10FF   00560          CALL    GETNUM
FE34 329EFE   00570          LD      (LINLEN),A
FE37 2174FF   00580          LD      HL,NDNTMS        ;GET # OF COLS
FE3A CD10FF   00590          CALL    GETNUM           ;FOR INDENTATION
FE3D 3251FF   00600          LD      (INDENT),A
FE40 2194FF   00610          LD      HL,LINMSG        ;GET # OF PRINTED
FE43 CD10FF   00620          CALL    GETNUM           ;LINES / PAGE
FE46 3236FF   00630          LD      (LINES),A
FE49 AF       00640          XOR     A                ;SET LINCNT
FE4A 322940   00650          LD      (LINCNT),A       ;AT TOP-OF-FORM
FE4D C32D40   00660          JP      SYSTEM           ;GO BACK TO SYSTEM
              00670 ;
FE50 CD50FE   00680 KBDMOD:  CALL    KBDMOD           ;CALL ORIG KYBD
FE51          00690 KEYSTR   EQU     KBDMOD+1         ;VECTOR LOCATION
FE53 FE01     00700          CP      BREAK            ;BREAK KEY?
FE55 C8       00710          RET     Z                ;YES
FE56 FE10     00720          CP      CTRLP            ;IS CHAR PRINT TOGGLE?
FE58 200A     00730          JR      NZ,TOPTST        ;NO, CHECK TOP OF FORM
FE5A 3A4FFF   00740          LD      A,(PRTFLG)       ;YES, TOGGLE PRINT
FE5D EE01     00750          XOR     1                ;FLAG STATUS
FE5F 324FFF   00760          LD      (PRTFLG),A       ;AND SAVE NEW FLAG
FE62 18EC     00770          JR      KBDMOD
FE64 FE14     00780 TOPTST:  CP      CTRLT
FE66 C0       00790          RET     NZ
FE67 3A2940   00800          LD      A,(LINCNT)       ;PARTIAL PAGE?
FE6A B7       00810          OR      A
FE6B C43EFF   00820          CALL    NZ,BOTTOM        ;YES, FORM FEED
FE6E 18E0     00830          JR      KBDMOD
              00840 ;
FE70 F5       00850 TVMOD:   PUSH    AF
FE71 C5       00860          PUSH    BC
FE72 D5       00870          PUSH    DE
FE73 3A4FFF   00880          LD      A,(PRTFLG)       ;FETCH PRINT FLAG
FE76 A7       00890          AND     A                ;TEST FOR ZERO
FE77 79       00900          LD      A,C
FE78 280D     00910          JR      Z,TVOUT          ;IF ZERO, DON'T PRINT
FE7A FE20     00920          CP      ' '              ;PRINTABLE CHAR?
FE7C 3006     00930          JR      NC,TVM0          ;YES
FE7E FE0E     00940          CP      CR+1             ;SCREEN CNTRL CHAR?
FE80 3802     00950          JR      C,TVM0           ;NO
FE82 0E0D     00960          LD      C,CR             ;YES, SUBST CR
FE84 CD8DFE   00970 TVM0:    CALL    PRNTR            ;PRINT CHAR
FE87 D1       00980 TVOUT:   POP     DE
FE88 C1       00990          POP     BC
FE89 F1       01000          POP     AF
FE8B          01010 TVSTR    EQU     $+1              ;JUMP TO ORIG
FE8A C30000   01020          JP      0                ;TV VECTOR LOCN
              01030 ;
FE8D F5       01040 PRNTR:   PUSH    AF
FE8E C5       01050          PUSH    BC
FE8F D5       01060          PUSH    DE
```

```
FE90 79       01070          LD      A,C             ;CHAR TO BE PRINTED
FE91 FE20     01080          CP      ' '             ;FORMS CNTRL CHAR?
FE93 3005     01090          JR      NC,NOFORM       ;NO
FE95 CD26FF   01100          CALL    FORMS           ;YES, HANDLE IT
FE98 182E     01110          JR      PRTOUT          ;EXIT
FE9A 3A50FF   01120 NOFORM:  LD      A,(CHRCNT)      ;FETCH CHAR COUNT
FE9D FE50     01130          CP      80              ;LINE FULL ALREADY?
FE9E          01140 LINLEN   EQU     $-1
FE9F 380A     01150          JR      C,CHK1ST        ;NOT YET
FEA1 C5       01160          PUSH    BC              ;FULL LINE
FEA2 0E0D     01170          LD      C,CR            ;PRINT CR
FEA4 CD26FF   01180          CALL    FORMS
FEA7 C1       01190          POP     BC
FEA8 3A50FF   01200          LD      A,(CHRCNT)      ;RESUME TESTING
FEAB A7       01210 CHK1ST:  AND     A               ;1ST CHAR IN LINE?
FEAC 2010     01220          JR      NZ,NOT1ST       ;NO
FEAE 3A51FF   01230          LD      A,(INDENT)      ;GET # TO INDENT
FEB1 B7       01240          OR      A               ;NO INDENT WANTED?
FEB2 280A     01250          JR      Z,NOT1ST        ;THAT'S RIGHT
FEB4 C5       01260          PUSH    BC              ;INDENT THE LINE
FEB5 47       01270          LD      B,A             ;# OF COLS TO B
FEB6 0E20     01280 INDLUP:  LD      C,' '           ;OUTPUT SPACES
FEB8 CDCCFE   01290          CALL    PRINT
FEBB 10F9     01300          DJNZ    INDLUP
FEBD C1       01310          POP     BC
FEBE CDCCFE   01320 NOT1ST:  CALL    PRINT           ;PRINT THE CHAR
FEC1 3A50FF   01330          LD      A,(CHRCNT)      ;UPDATE THE CHAR COUNT
FEC4 3C       01340          INC     A
FEC5 3250FF   01350          LD      (CHRCNT),A
FEC8 D1       01360 PRTOUT:  POP     DE
FEC9 C1       01370          POP     BC
FECA F1       01380          POP     AF
FECB C9       01390          RET
              01400 ;
FECC F5       01410 PRINT:   PUSH    AF
FECD E5       01420          PUSH    HL
FECE C5       01430          PUSH    BC
FECF 015000   01440          LD      BC,80           ;SET UP TIME OUT VALUE
FED2 3AE837   01450 PR1:     LD      A,(PTSTAT)      ;GET PRINTER STATUS
FED5 E6F0     01460          AND     0F0H
FED7 FE30     01470          CP      30H             ;PRINTER READY?
FED9 2811     01480          JR      Z,PR2           ;YES, GO TO IT, KID!
FEDB 10F5     01490          DJNZ    PR1             ;NO, CONTINUE TIMEOUT
FEDD 0D       01500          DEC     C
FEDE 20F2     01510          JR      NZ,PR1
              01520 ;
              01530 ; IF IT GOES ALL WAY THRU TIMING LOOP, PRINTER
              01540 ; EITHER IS NOT TURNED ON OR ISN'T IN THE SYSTEM!
              01550 ;
FEE0 97       01560          SUB     A               ;RESET PRINT FLAG
FEE1 324FFF   01570          LD      (PRTFLG),A      ;TO TV ALONE
FEE4 21B4FF   01580          LD      HL,NOPRNT       ;SEND ERROR MSG
FEE7 CDFFFE   01590          CALL    LINOUT
```

```
FEEA 180F      01600          JR      PRTRET          ;AND EXIT
FEEC C1        01610 PR2:     POP     BC              ;RETRIEVE ORIG CHAR
FEED C5        01620          PUSH    BC              ;AND RESAVE IT
FEEE 79        01630          LD      A,C             ;FETCH CHAR TO PRINT
FEEF 32E837    01640          LD      (PRINTR),A      ;FOR MODEL I ONLY
               01650 ;        OUT     (0F8H),A        ;FOR MODEL III ONLY
FEF2 3AE837    01660 WAITLP:  LD      A,(PTSTAT)      ;WAIT FOR PRNTR RDY
FEF5 E6F0      01670          AND     0F0H            ;BEFORE LEAVING
FEF7 FE30      01680          CP      30H
FEF9 20F7      01690          JR      NZ,WAITLP
FEFB C1        01700 PRTRET:  POP     BC
FEFC E1        01710          POP     HL
FEFD F1        01720          POP     AF
FEFE C9        01730          RET
               01740 ;
               01750 ; DISPLAYS MESSAGE ON SCREEN
               01760 ; HL REGS PNT TO 1ST CHAR OF MSG
               01770 ; MSG MUST HAVE 00 AS LAST BYTE
               01780 ;
FEFF F5        01790 LINOUT:  PUSH    AF
FF00 E5        01800          PUSH    HL
FF01 7E        01810 L1:      LD      A,(HL)
FF02 23        01820          INC     HL
FF03 B7        01830          OR      A
FF04 2807      01840          JR      Z,L2
FF06 D5        01850          PUSH    DE
FF07 CD3300    01860          CALL    TV
FF0A D1        01870          POP     DE
FF0B 18F4      01880          JR      L1
FF0D E1        01890 L2:      POP     HL
FF0E F1        01900          POP     AF
FF0F C9        01910          RET
               01920 ;
FF10 C5        01930 GETNUM:  PUSH    BC
FF11 D5        01940          PUSH    DE
FF12 E5        01950          PUSH    HL
FF13 CDFFFE    01960          CALL    LINOUT
FF16 061E      01970          LD      B,30            ;GET DECIMAL LINE LEN
FF18 2100FE    01980          LD      HL,START
FF1B CD4000    01990          CALL    LINEIN
FF1E CD4F1E    02000          CALL    DECBIN          ;CNVRT TO BINARY
FF21 7B        02010          LD      A,E             ;MOVE TO ACC
FF22 E1        02020          POP     HL
FF23 D1        02030          POP     DE
FF24 C1        02040          POP     BC
FF25 C9        02050          RET
               02060 ;
FF26 79        02070 FORMS:   LD      A,C             ;FETCH CHAR TO PRINT
FF27 FE0C      02080          CP      FF              ;FORM FEED CHAR?
FF29 2813      02090          JR      Z,BOTTOM        ;YES, END PAGE
FF2B CDCCFE    02100          CALL    PRINT           ;NO, PRINT CHAR
FF2E 3A2940    02110          LD      A,(LINCNT)      ;UPDATE LINE COUNT
FF31 3C        02120          INC     A
```

```
FF32 322940    02130          LD      (LINCNT),A
FF35 FE36      02140          CP      54                ;FINISHED W/ PAGE?
FF36           02150 LINES    EQU     $-1
FF37 3005      02160          JR      NC,BOTTOM         ;YES, END PAGE
FF39 AF        02170          XOR     A                 ;NO, RESET CHAR COUNT
FF3A 3250FF    02180          LD      (CHRCNT),A
FF3D C9        02190          RET
               02200 ;
FF3E C5        02210 BOTTOM:  PUSH    BC
FF3F D5        02220          PUSH    DE
FF40 0E0C      02230          LD      C,FF
FF42 CDCCFE    02240          CALL    PRINT
FF45 AF        02250          XOR     A                 ;INIT LINCNT
FF46 322940    02260          LD      (LINCNT),A
FF49 3250FF    02270          LD      (CHRCNT),A        ;INIT CHRCNT
FF4C D1        02280          POP     DE
FF4D C1        02290          POP     BC
FF4E C9        02300          RET
               02310 ;
               02320 ; PRINT FLAG STORAGE: DEFB IS USED TO INITIALIZE
               02330 ; TO 'OFF' STATUS
               02340 ;
FF4F 00        02350 PRTFLG:  DEFB    0
               02360 ;
FF50 00        02370 CHRCNT:  DEFB    0
FF51 00        02380 INDENT:  DEFB    0
               02390 ;
FF52 45        02400 LENMSG:  DEFM    'ENTER MAX NO. OF CHARS PER LINE: '
FF53 4E
FF54 54
FF55 45
FF56 52
FF57 20
FF58 4D
FF59 41
FF5A 58
FF5B 20
FF5C 4E
FF5D 4F
FF5E 2E
FF5F 20
FF60 4F
FF61 46
FF62 20
FF63 43
FF64 48
FF65 41
FF66 52
FF67 53
FF68 20
FF69 50
FF6A 45
FF6B 52
```

```
FF6C 20
FF6D 4C
FF6E 49
FF6F 4E
FF70 45
FF71 3A
FF72 20
FF73 00         02410           DEFB    0
                02420 ;
FF74 45         02430 NDNTMS: DEFM      'ENTER NO. OF SPACES TO INDENT: '
FF75 4E
FF76 54
FF77 45
FF78 52
FF79 20
FF7A 4E
FF7B 4F
FF7C 2E
FF7D 20
FF7E 4F
FF7F 46
FF80 20
FF81 53
FF82 50
FF83 41
FF84 43
FF85 45
FF86 53
FF87 20
FF88 54
FF89 4F
FF8A 20
FF8B 49
FF8C 4E
FF8D 44
FF8E 45
FF8F 4E
FF90 54
FF91 3A
FF92 20
FF93 00         02440           DEFB    0
                02450 ;
FF94 45         02460 LINMSG: DEFM      'ENTER NO. OF PRINT LINES/PAGE: '
FF95 4E
FF96 54
FF97 45
FF98 52
FF99 20
FF9A 4E
FF9B 4F
FF9C 2E
FF9D 20
FF9E 4F
```

```
FF9F 46
FFA0 20
FFA1 50
FFA2 52
FFA3 49
FFA4 4E
FFA5 54
FFA6 20
FFA7 4C
FFA8 49
FFA9 4E
FFAA 45
FFAB 53
FFAC 2F
FFAD 50
FFAE 41
FFAF 47
FFB0 45
FFB1 3A
FFB2 20
FFB3 00          02470          DEFB    0
                 02480 ;
FFB4 0D          02490 NOPRNT:  DEFB    CR
FFB5 50          02500          DEFM    'PRINTER NOT READY'
FFB6 52
FFB7 49
FFB8 4E
FFB9 54
FFBA 45
FFBB 52
FFBC 20
FFBD 4E
FFBE 4F
FFBF 54
FFC0 20
FFC1 52
FFC2 45
FFC3 41
FFC4 44
FFC5 59
FFC6 0D00        02510          DEFW    CR
                 02520 ;
FE00             02530          END     START
00000 Total Errors

L2       FF0D
L1       FF01
WAITLP   FEF2
PRTRET   FEFB
LINOUT   FEFF
NOPRNT   FFB4
PR2      FEEC
PR1      FED2
```

CHAPTER 5
FOLLOW THE BOUNCING BALL

One problem that many novice assembly language programmers have is putting the instructions they have laboriously studied together to make a program. It is akin to studying a dictionary to learn the definitions of many words, but then finding out that you do not know how to form sentences, paragraphs, chapters, etc.

If you are going to be able to author that veritable masterpiece of a program, you must learn how to string instruction sequences together to accomplish something worthwhile. We have seen a couple of examples of programs already, and I have tried to give you some idea of WHY I chose the given instruction sequences. However, the examples were fairly short, though useful.

What I want to do now is to present a rather lengthy program to you. I will attempt to give you a perspective of how an experienced assembly language programmer (you guessed him -- me!) approaches a non-trivial programming task. You may be surprised to find it very similar to the approach you have developed in your BASIC or other programming.

I cannot resist, and indeed have not resisted, the opportunity to interlace liberally my opinions, biases, and sggestions about general techniques of programming throughout the discussion of the details pertinent to the present example. So what follows is a mixture of general and specific, with my goal being for you to thoroughly understand what I did and why I did it. Keep in mind that what I will be demonstrating is not THE right way to do a given task, but only ONE OF THE MANY "right" ways!

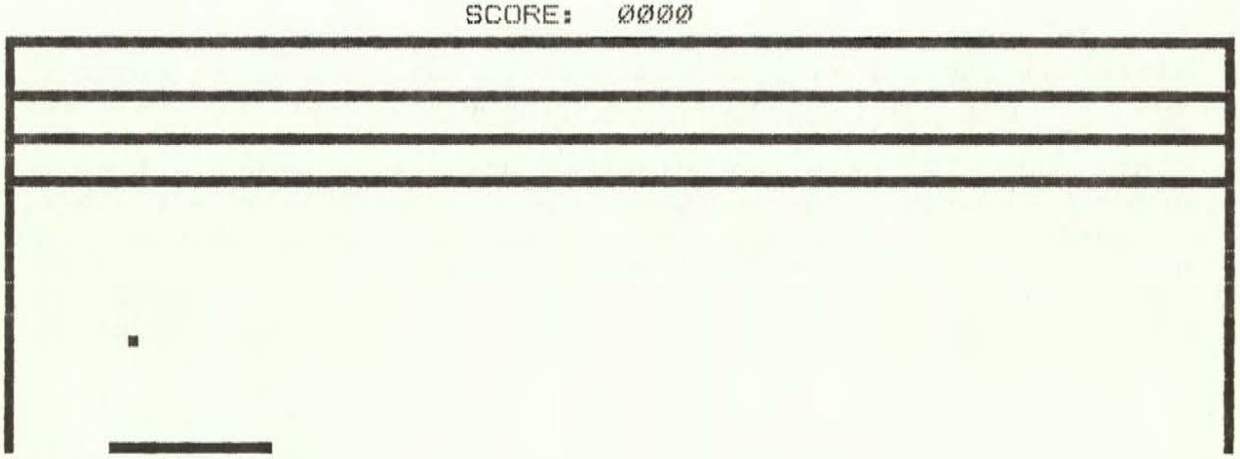
It will require careful study on your part to assimilate all that I will be discussing. It also will require you to key the program in and assemble it if you want to get maximum benefit from the exercise. To make it truly fun, you will have to implement some enhancements I will suggest later! Remember, learning assembly or any other language is not achieved by merely reading or listening to a lecturer! You must practice!, practice!, practice!, practice!, and then practice your practice! I wish there were some easier way, but you are a friend and I must be truthful with you!

ON TO THE PROGRAM!!! First, what is it? It is a game! Now don't elevate your nose in snobbery!! Games programming is worth anyone's attention, even though you intend to do something entirely different later on with your knowledge of assembly programming. Many of the techniques for getting fast action on the screen are remarkably similar to techniques for acquiring data in "real time" from an A/D converter, etc. So, listen up!

What is the game? Its name is BOUNCE, a fact which should give you a clue as to its nature. You guessed it!! It is a ping-pong-type game with a ball and paddle, and three walls to bounce the ball within. To make it a little more interesting and to show some neat techniques, I have added three rows of blocks for the ball to hit. A hit on any block will increment your score, the magnitude of the increment depending on which row of blocks the ball strikes. Before the game starts, you must input a skill level from 1 through 9, with 9 the easiest and slowest play. When you miss a ball with the paddle (a common occurence with me, I am sorry to say!), the program will prompt you with a "new ball" message. So it is very forgiving of misses! You get an unlimited number of balls.

If you are familiar with some of the commercial games which involve bouncing balls, you will quickly see that BOUNCE is a primitive relative at best! However, I cannot think of a better way to illustrate many of the advantages of assembly language programming in such a graphic (pun intended) manner. The purpose of the exercise is TUTORIAL, so it will be to your advantage to study it closely, even though you are not the least bit interested in the game as such.

The playfield looks like this:



The TRS-80 text screen is made up of 16 rows of 64 characters each. As usual when dealing with computers, the numbering starts with zero, so the vertical rows are numbered from 0 through 15 and the horizontal columns range from 0 and 63, inclusive. The top of the screen is row 0; the bottom is row 15. The left-most column is column 0; the right-most is column 63.

For graphics, each row is further split into three zones and each column is split into two zones (details start on page 5-6), so we have a total of 48 vertical positions (0 through 47) and 128 horizontal positions (0 through 127). In keeping with mathematical convention, I have called a vertical position a "Y" coordinate and a horizontal position an "X" coordinate. Thus, the position of the ball anywhere on the screen can be described by its X and Y coordinates. In the BOUNCE program I have called these "X" and "Y" -- no need getting tricky here!

So we know how to describe the ball's position. But, since the ball is moving, there are two other things we need to describe. First, we need to know in which direction the ball is traveling. The direction has two components, one for each coordinate. The X direction (XDIRXN) is positive if movement is from left to right, in other words, from smaller to larger X values; it is negative when movement is from right to left. The Y direction (YDIRXN) is positive when movement is from top to bottom, corresponding to increasing Y values; it is negative when movement is upward on the screen, from bottom to top.

Secondly, we need to know how much movement there is in each direction, i.e., the changes in X position and Y position, in the process of each "main move" of the ball. Once again in keeping with mathematical nomenclature, I call these XDELTA (for the change in X) and YDELTA (for the Y change). Both of these values have allowable amounts of 1 through 3, as you will see in detail later.

BACKGROUND INFORMATION

Before we start batting around in the game, I want to make sure that you and I know the same things about our TRS-80s concerning keyboard and screen functions. Of particular interest is the fact that both the keyboard and the screen are "memory-mapped". Well, Willis, what does that mean, you impatiently ask?! Let me digress from this digression to discuss the concept of memory-mapping.

The Z80 chip in our favorite computer is capable of addressing 64K of memory. The actual amount is 65,536 bytes. In addition to that, as I said in one of the lectures (you HAVE already listened to ALL the lectures, haven't you???), the Z80 can access 256 input ports and 256 output ports. So designers of Z80 computer systems have the choice of making any input/output device appear as one or more I/O ports OR ONE OR MORE MEMORY LOCATIONS! An overwhelming fact about our smart CPU is that it is dumb! We can very easily fake it into thinking that a byte of data it fetches from a given address is from an ordinary memory chip, when in fact it may be from contact closures on a keyboard, from an analog to digital converter, from a temperature sensor, from a joystick, etc.

Exactly how this is done is certainly beyond the scope of REMASSEM, but that does not prevent us from using the information. In this case, the keyboard is set up to use locations from 3800H to 3880H in memory. The actual configuration is not important to our discussion here. A full schematic of the keyboard addressing is in the pertinent Radio Shack Technical Reference Handbook for your TRS-80 model. If you do much assembly language programming, you will find purchase of the handbook worth its cost.

What I wanted to do was use the left arrow and right arrow keys on my TRS-80 to control the paddle. I did not have the technical handbook at hand, so I looked at addresses 3800H to 3880H with DEBUG and saw what address or addresses changed values when I pressed either the left arrow or the right arrow. This is a cumbersome way to work -- in fact it is nigh onto impossible with a Model III -- it is much better to use the handbook! It did have one beneficial result; it reminded me to tell you what follows. You will find if you try this that several addresses change -- this is a result of incomplete decoding of the addresses. An analogy would be that your home address is given just as a street in the city, rather than as a house on a street within a city. I admit it is an imperfect analogy since, in the keyboard's case, any address which shows the change will work. Fear not, just pick any one of the addresses you prefer and forge ahead! Any one will work as well as any other. The other residents on your street may well be irritable if you try the same thing with residences!

The left arrow and right arrow keys are located at 3840H (among other addresses) and this address holds 20H (32 decimal) when the left arrow is the one pressed and 40H (64 decimal) when the right arrow is pressed. So we can monitor for our desired keys just by looking at the value shown at one address!

What does this gain us? Why not just use the keycheck routine in the ROM? The keycheck routine (located at 2BH) returns a zero in the accumulator if no key is pressed and returns the ASCII value of the key if one is pressed. The problem with this arises if you would like to keep from pressing a key repetitively, for example, to move the paddle in our BOUNCE program. The keycheck routine requires you to release the key and re-press it each time. Keep that in mind, since some times that is exactly what you want -- the keycheck routine is a useful one.

However, any program should be "user-friendly". That is a very overworked term, but very true! No one will want to play a game for long if the left- and right-arrow keys must be quickly and violently pressed, released, pressed, released, etc., to move the paddle under the bouncing ball. No computer owner wants to see the keyboard mistreated in such a fashion, either! The BOUNCE program will illustrate how you can monitor a particular key or group of keys, and continue a desired action for as long as the key is pressed.

That is enough about the keyboard for now. What about the screen? Screen memory for both Models I and III resides in the area between 3C00H and 3FFFH, inclusive. When you invoke the ROM routine (at 33H) to display a character on the screen, it picks up a pointer to the next address to receive a character (the so-called "cursor position"), places the character there, updates the cursor position, and returns to the caller. When appropriate, the ROM routine effects scrolling, clears the screen, erases to the end of the current line, etc.. The pertinent point is that what is displayed on the screen is under software control. The hardware circuitry is continually and automatically displaying whatever resides in the screen memory area.

It surely has occurred to you that if the ROM routine can write to a specific address, so can you and I. We can also read the value at a specific address. This gives us true random access to any character on the screen!

Why Memory-Mapping?

What advantages do we receive from memory mapping of an input/output device to compensate for the reduced amount of RAM we then have available for general use? After all, we have those 256 input ports and 256 output ports available to us and we surely could use more RAM memory. (I have often wondered whether the programmers of mainframe computers having megabytes of RAM feel as you and I do -- there is almost but not quite enough memory to do the job we want! There seems to be some natural law to this effect!)

In the case of the keyboard, I cannot help but feel that the main consideration of the TRS-80 designers was COST! Whether or not this is true, a memory-mapped keyboard offers many advantages to the programmer. An excellent example is our use of the left and right arrow keys in the BOUNCE program. This likely would not be possible if a hardware keyboard decoder had been designed into our favorite computer. Furthermore, software decoding of the keyboard allows simultaneous multiple key depressions to have special meaning. Examples of this are the Model III screen print function and the NEWDOS screen print function invoked by simultaneous depression of the J, K, and L keys. I am sure you have seen other examples in programs you have studied. Software decoding offers a clever programmer a very versatile input system.

Concerning the screen, memory mapping provides quick access to any portion of the screen. This is why the most useful PRINT@ function has been implemented in the Models I and III versions of Microsoft BASIC. It is a simple matter for the BASIC interpreter to take the number following the "@", add this to the start of screen memory, and place this address in the cursor position pointer referred to above. The next character to be displayed will show up at this location on the screen.

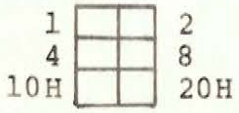
An even more useful result of a memory-mapped screen, at least in our BOUNCE task, is that a particular location on the screen can be accessed and inspected. This forms the basis of the graphics capability in BASIC -- the SET, RESET, and POINT functions. As you already know, the Models I and III screens have 16 lines of 64 characters each. For graphics the screen is divided into a matrix of 48 vertical locations and 128 horizontal locations. I will call these smallest units "pixels".

Now for some quick calculations. Screen memory stretches from 3C00H through 3FFFH; that is 400H or 1024 bytes. In the graphics mode we have 48 times 128 locations which calculates to 6144 pixels. So each byte of screen memory has <u>six</u> graphics pixels within it. This means you and I will have to calculate more than just the byte address when we wish to SET, RESET, or POINT. How can we do this?

Below is the graphics character set. The first item shown in each group is the graphics character itself; next is the decimal value of that character; third is the corresponding hexadecimal value; and last is the hex value after the graphics bias of 80H has been removed.

▘	129	81	01	▝	130	82	02	▀	131	83	03	▖	132	84	04
▌	133	85	05	▞	134	86	06	▛	135	87	07				
▗	136	88	08	▚	137	89	09	▐	138	8A	0A	▜	139	8B	0B
▄	140	8C	0C	▙	141	8D	0D	▟	142	8E	0E				
▉	143	8F	0F		144	90	10	⠁	145	91	11	⠂	146	92	12
⠃	147	93	13	⠄	148	94	14	⠅	149	95	15				
⠆	150	96	16	⠇	151	97	17	⠈	152	98	18	⠉	153	99	19
⠊	154	9A	1A	⠋	155	9B	1B	⠌	156	9C	1C				
⠍	157	9D	1D	⠎	158	9E	1E	⠏	159	9F	1F		160	A0	20
⠑	161	A1	21	⠒	162	A2	22	⠓	163	A3	23				
⠔	164	A4	24	⠕	165	A5	25	⠖	166	A6	26	⠗	167	A7	27
⠘	168	A8	28	⠙	169	A9	29	⠚	170	AA	2A				
⠛	171	AB	2B	⠜	172	AC	2C	⠝	173	AD	2D	⠞	174	AE	2E
⠟	175	AF	2F	⠠	176	B0	30	⠡	177	B1	31				
⠢	178	B2	32	⠣	179	B3	33	⠤	180	B4	34	⠥	181	B5	35
⠦	182	B6	36	⠧	183	B7	37	⠨	184	B8	38				
⠩	185	B9	39	⠪	186	BA	3A	⠫	187	BB	3B	⠬	188	BC	3C
⠭	189	BD	3D	⠮	190	BE	3E	⠯	191	BF	3F				

Most of the characters shown are combinations of the six fundamental characters. The six unique characters are 81H, 82H, 84H, 88H, 90H, and 0A0H. By inspecting the hex values after removal of the graphics bias, we see the following relationship:



So if we wanted to turn on only the upper left pixel and the lower right pixel of a given byte, we can calculate the value of the byte we send to the screen by first adding 1 (for upper left) and 20H (for lower right) and then adding the graphics bias, 80H, getting a final result of 0A1H. When we look up 0A1H (161 decimal) in the above table we find the expected graphics representation. So we now have the information we will need later for our assembly language SET, RESET, and POINT functions.

PROGRAMMING PREFERENCES, PREJUDICES, AND PRACTICES

I threatened to give you my opinions, biases, etc, about programming, so now is as good a time as any! My experience has been that the most common problems with programmers in ANY language is that they do not spend enough time planning and designing a program. I touched on this in Lesson 10. Although professional programmers are less prone to rush to the coding phase (perhaps!), it is almost universal among amateur microcomputer programmers to proceed quickly to the computer to begin to write instructions. After all, the computer is just sitting there waiting for me! I do not have to "time-share" with anyone else, and no one is looking over my shoulder to see my mistakes. What's the harm?

Do not let me burst your bubble, but an excellent way to judge the experience and acumen of a programmer is to observe how much time elapses and how it is spent between the initial consideration of the task and the beginning of the actual coding. My personal experience has been that the time spent designing a program, considering alternative algorithms or data structures, etc., is more than compensated for by the decreased debugging and rewriting time.

Once I have the structures of the program and data well in hand, I begin to write code. The first code written usually is the main routine with many subroutine calls; I'll say more about subroutines later. Then the various subroutines are written, with frequent testing during this process. Subroutines not yet written have just a RET instruction, i.e., they are simple "stubs". Furthermore, subroutines when first written are often mere "skeletons". Much flesh will be added before they are complete. I routinely follow the process of stepwise refinement I mentioned in Lesson 10. In this way, you can get the program "off the ground" fairly quickly. This is particularly important in routines which must operate in rapid, real-time fashion. You can judge the time limitations much better when you have a good, though simple, program operational. You may have to make compromises or revisions in your original algorithms, based on the performance of your program; the sooner you determine this, the less time and coding you waste. After you get initial performance data, then you can go back to augment and refine individual subroutines as well as code other "stub" routines for their initial testing.

In this manner the program becomes gradually more intricate. Poor algorithm choices will show up early, before you are tempted to patch the existing program because you have already invested so much time. This patching seldom works out, anyway, so I have found stepwise refinement one of the most useful approaches to program development.

All of this sounds simply grand, but we both know that there are times when you get a "brainstorm" well after program coding has begun, requiring a change of algorithm or at the very least a modification of some of the existing code. Modular programming has much to recommend it in this case. More about this later.

Let's look first at an overview of BOUNCE. There are three main features which I want to bring to your attention:

1. Many EQUate statements.

2. Many commented instructions.

3. Many subroutines.

All three of these practices are commonly followed by me when I am writing anything other than a trivial, short, quick and dirty routine. Even there, I have been puzzled upon looking back at routines I wrote a year or two ago!! Now what did I have in mind when I wrote that routine? A few comments and equates would be quite useful NOW!!

Although having a large program broken up into several subrutines with liberal equates and comments can be very helpful in later revisions or reviews, I feel that these features are even more important when you are first writing the program. Let me give some reasons for this statement, based on my experience.

As you have probably guessed by now even if you have not had much first-hand experience, efficient use of assembly language in many cases requires considerable knowledge of the hardware configuration of our computer. Acquisition of this information takes many paths, such as study of technical manuals and schematics, magazine articles, disassembly of commercial programs, etc. It is most important when we incorporate these items in our own programming that we choose meaningful mnemonics as soon as possible, so that we do not have to remember all of the details.

An example, on line 110 of the BOUNCE program, shows KEYCHK equated to 2BH. This is the keycheck routine discussed earlier. Later in the program when we see a

```
CALL  KEYCHK
```

statement we likely will understand what is to be done by the routine.

This may not be true if we encountered a

```
CALL 2BH
```

statement. So an obvious use of the EQU pseudo-op is to define constants, so that they can be referred to symbolically for ease of use. This is illustrated in lines 110-180 of BOUNCE. As long as we work with the same computer, these values will not change.

Another obvious advantage of the EQU pseudo-op is illustrated in lines 190-260 of BOUNCE. Here once again we are defining constants (the EQU statement ALWAYS defines a constant), but we may want to change these constants and reassemble to get differing effects. In the present case, I am defining the rows and columns which make up the playfield boundaries. The screen has 16 rows (numbered 0 through 15), each of which is comprised of 64 characters (numbered 0 through 63). So here I am saying that the top wall of the playfield will be in row 1 (the second from the very top); the left wall is in column 0 and the right wall is in column 63. The left and right walls extend through row 13. As you will see later, this is also where the paddle will be located.

If you always want the playfield's dimensions to be the same, you may be tempted to use the actual values in the pertinent instructions (e.g., in line 560 or line 620.). Doing this, however, would force you to remember the significance of the number, probably requiring you to insert a specific comment in the source code. Also, if you add a feature later on which requires a change in the playfield dimensions, you will have to scan the program carefully to change all appropriate values to the new values. It is much simpler and more efficient to change the corresponding equates in lines 190-220.

An overview of the BOUNCE program reveals that it has many modules. I am a firm believer in the benefits of modular programming. In fact you will find some subroutines in BOUNCE that are called only one time! Conventional wisdom says that you should count bytes to determine whether you split out the code as a subroutine or place it "in-line" for the times you need it. Obviously, the more times an instruction sequence is needed in a given program, the more desirable it is to make that sequence a subroutine. Certainly a routine that is called only once is wasteful of memory (for the CALL and RET instructions) as well as time (for the CALL and RET instructions). So why have a subroutine? Well I admit that I normally move the code in-line after I am sure that I need it only once. I deliberately left the routines in BOUNCE as is to illustrate a point.

My point is that anything I can do to limit the amount of information I need to keep in mind when writing any particular section of code is most helpful. If you consider breaking up a complex task into several simple tasks you will be well ahead. So I think in terms of subroutines which each have only one or two functions. Furthermore, I ordinarily follow the practice of having the subroutine preserve all registers except the accumulator; the sole regular exception to this is when the subroutine must pass data back to the calling routine using one or more registers. When I am coding a subroutine, I need only to remember what the subroutine will need as data or will receive as input, what the subroutine must do to or with these data, and what output must issue as a result of the subroutine's action. The subroutine can be a true "black box" to the calling routine, which does not need to know anything about how the subroutine works! If the program is well designed, I have no worry about strange interaction of subroutines.

This is why the design step is SO important! This is when you must consider the interaction of the modules -- in fact, this should be the ONLY time you consider the hierarchy of the various modules. If you must keep these facts in mind throughout the coding process, you have not really modularized the program! Shame on you! Go back and redo it until you get it right.

DISSECTION OF BOUNCE

I want now to go into a step-by-step discussion of the BOUNCE program. A complete assembly listing of the program starts on page 5-31. If you have not already done so, I suggest you take the listing out of the binder and place it alongside the text so that your eyes can move readily from one to the other. For lack of a better approach, I will discuss the modules in the order they occur in the listing. The number(s) after the heading of each section below is the line number(s) of the first statement in the source code segment.

Equates (110)

Quite a bit has already been said about the equates, so I will not spend much more time here. Just refer back to these as you need to during the discussion of the various routines. The only thing I think I need to point out at this time is the use of the "shift" pseudo-op, as illustrated in line 230 of BOUNCE. The "<" means to perform an arithmetic shift -- if the number following the "<" is positive, it is a shift left; if negative, a shift right. In line 230 I am shifting left 1 bit (equivalent to multiplying by 2) and then adding the original value + 1 to it. The net effect is ROWMIN * 3 + 1. Likewise, in line 260 I am multiplying COLMAX * 2. The reasons for this will become apparent as we discuss the routines using these values. Remember that a pseudo-op is an instruction to only the assembler. The assembler will do the calculations if an expression is given; the result of this expression evaluation is a constant.

Main Routine (400)

The main routine extends from line 400 through line 1400. We first must clear the screen and initialize the score to zero. As you have seen earlier, there is a ROM routine at 1C9H which nicely clears the screen; I did not use that one for reasons I'll discuss later.

Next, in lines 440-460, the "SCORE:" message is displayed on the screen. Then the top line of the playfield is drawn. The segment in lines 470-510 takes row and column data (ROWMIN and COLMIN, in this case) and converts that into a specific address in screen memory; I'll say more about the GETADR routine later. Line 520 loads the accumulator with 97H, the graphics character for the top left corner of the playfield (see above for what the character looks like), and stores it in the location pointed to by the HL register pair. After incrementing the memory pointer in line 540, consecutive 83H's are written across the screen and finally a 0ABH forms the top right corner of

the playfield. Note the use of register B as a loop counter, using the DJNZ instruction discussed in the audio lectures.

In like manner, the left and right walls are drawn in lines 620-710. The paddle is then drawn in lines 720-780. Since the paddle can be (and will be, eventually) drawn anywhere on the appropriate line across the screen, I use a subroutine (PDLSET) which needs only the left-most paddle location and the width of the paddle in pixels. Note the use of the accumulator to pass the position to the PDLSET routine. Since the subroutine changes the value in the accumulator, we must preserve it here with PUSH and POP instructions. Remember, the POP instruction will duplicate the two bytes pointed to by the stack pointer into the 16-bit register or register pair indicated in the instruction. The stack pointer will then be incremented by two. Details, details!! Actually, the stack pointer picks up one byte at a time, incrementing after each byte, but the net effect is the same -- the stack operations are all 16-bit operations. You cannot just POP B, you must POP BC.

It is important to remember that there is nothing magic about the stack pointer or the stack itself. The stack is ordinary RAM just like any other. Why is this important to remember? Simply this -- you can have a whole series of POPs without corresponding PUSHes; you will never exhaust the stack -- it will keep on giving you 16 bits of data each POP, long after you have gotten all of the meaningful information. The programmer MUST match PUSHes and POPs! Otherwise, it's GIGO!!! (Garbage In, Garbage Out). The garbage your program is eating may be itself!

Line 790 calls the routine to draw the blocks. More about this later.

Lines 800-960 prompt for the desired skill level of play, convert the ASCII digit obtained to a binary number, store it away for later use, and erase the skill level message. WHEW! Actually, it is not so difficult. First we display the prompt message and call for a character from the keyboard. Lines 840-870 insist that it be a digit between 1 and 9, inclusive. These instructions show very typical uses of the flags to determine appropriate action. The carry flag after a compare operation is set only if the value in the accumulator is less than the value with which you are comparing it. The four RLCA instructions are an economical (4 bytes) way to multiply an 8-bit value by 16. Later on you will see a way to divide by 16.

Lines 940-960 may seem a crude way to erase a message from the screen, but it really is not. You will find many of the operations in assembly language rather primitive, but effective!

Line 970 labelled "NUBALL" resets the stack pointer to the same address given in line 400. Why is this instruction necessary? Any time you want to restart a section of your program and want to be sure you have a balanced stack, you should use this kind of instruction. So, what is a "balanced" stack, Willis? If you have a corresponding POP for every PUSH, the stack should remain in balance. Likewise you should match up CALLs and RETurns. Keep in mind that when a RET instruction is encountered, the next two bytes on the stack will be popped into the program counter and execution will begin there. If data PUSHed by a subroutine are still on the stack (not POPped off), the top two bytes of these data will be used as the return address. That will produce INTERESTING behavior, but probably not DESIRABLE behavior! A stack is considered balanced if there is a balancing POP for every PUSH, a balancing RETurn for every CALL, etc.

Why would the stack not be balanced at this point? Obviously, the first time through it would be balanced! NUBALL is a re-entry point, however, as you will see later, and can be re-entered from different levels of nested subroutines. The easiest way to clean things up is to simply reset the stack pointer to a safe value. In this example, I am writing a stand-alone program, so I know where a safe area of memory is for the stack. In many situations I will be interfacing with unknown other programs and will not know an absolute address which is all right for the stack pointer. In those cases, line 400 could be replaced by:

```
LD  (STAK),SP
```

where STAK is a 16-bit data area somewhere in your program. For example, you could place it just after the SCORE data area shown in line 5810. Then line 970 could be:

```
NUBALL   LD  SP,(STAK)
```

Although you do not know ahead of time where the stack pointer is pointing, it should be apparent that you no longer need to know.

A register you probably will not use too often is used in line 980. The refresh register, R, is a very nice feature of the Z80 chip; it allows use of so-called "dynamic" memory chips. I do not really know why they are called dynamic, and why the other kind is called static, but the pertinent characteristic is that dynamic cells lose their memory if not refreshed quite often. WHO SAYS COMPUTERS ARE NOT HUMAN-LIKE?? The Z80 chip, using the R register as the lower 8 bits of the address, performs this refresh operation. The merely mortal programmer has enough to worry about without this, so the chip takes care of it all by itself -- the jargon is "the refresh operation is 'transparent' to the programmer". Since the R register is being continually incremented by the Z80 chip, it can serve as a pseudo-random number source for a program. Let us discuss random numbers for a while.

For a number to be truly random, there must be something which has occurred that was "random". An example is any interaction between a human and the computer. The program is not at all random in nature -- each instruction takes a given number of clock cycles, and if one needs to, one can calculate the number of clock cycles a sequence of instructions will take to execute. On the other hand, the human is very unpredictable as far as how long he or she will take to respond to a prompt string, for example, and so an element of randomness can be introduced in this way.

In the same manner, the precise time that a program begins execution is random because of the human. So this first use of the R register will give at least a pseudo-random value. On the other hand, the use of the R register in line 1220 will not give a random number because there is a fixed number of clock cycles between the instruction in line 980 and the instruction in line 1220. It does not matter for our purposes here, but the concept should be understood well, since you will need random numbers randomly in the future.

After we get the pseudorandom number, we must test it to make sure it lies in value between XLOW and XHIGH. Once again, in lines 1000-1050, we use the carry flag information to advantage. The Y location is always just under the lower row of blocks.

Next we need to get values for XDELTA and YDELTA, which are the number of pixels of movement of X and Y, respectively, in each "main move" of the ball. This is done to make the ball movement less predictable -- how much fun would a game be if the ball always bounced in the same way, at the same speed? Values of XDELTA and YDELTA can be between 1 and 3, the way I have it set up. Here is a good part of the program for you to play around with. In lines

1090-1200, I get a little tricky. First I get the "random" number and save it by pushing it on the stack. Then I AND the number with 3, which will limit XDELTA values to the range 0-3. Since I do not want a 0 value for the number of pixels locations moved in the X direction, I test for 0 in line 1120 and replace a 0 with 1. So far there is nothing tricky.

Here's the tricky part. In line 1100 we pushed AF to preserve the original value we got from the refresh register. In line 1150 the value is retrieved from the stack and compared with 3FH. Since the number in the R register will vary between 0 and 7FH in the TRS-80 system, 3FH is the midway point; there is equal probability of the number being less than or greater than this value. So in effect this is a coin toss type of situation. From this information, I can end up with either a -1 or a +1 in the accumulator, as shown in lines 1160-1200. This value is stored in XDIRXN -- a +1 will cause movement from left to right, initially. A -1 will cause movement of the ball from right to left, initially.

We get another "random" number to determine YDELTA in the same way we did for XDELTA. YDIRXN is always +1 initially, since we are starting the ball out just under the lowest row of blocks and want it to come downward toward the paddle eagerly awaiting it.

We are now ready to show the initial position of the ball; we do this in line 1290 by a call to SETXY. Now we are down to the heart of the main routine. Lines 1300-1380 form a loop which is executed repetitively. As you can see, almost the whole loop is a series of CALLs to subroutines. The first one called is MUVBAL, which moves the ball, logically enough! Then we waste some time -- this is a major difference between BASIC and assembly language -- in BASIC the normal need is to speed things up; in assembly language you often must slow things down so that the "slow" human can see and react to the action!

We then update the score, if necessary, by a call to TVSCOR in line 1320. Lastly, we check to see if a key is pressed, and if so is it a space character? Note the use of the KEYCHK routine in ROM. It performs like the INKEY$ function in BASIC, in that it does not stop the action. If no key is pressed, the routine returns a zero, else the ASCII value of the particular key which was pressed. This last section, lines 1330-1380, was placed in the main routine to facilitate testing and debugging of the program. After bouncing the ball around for a while, I often wanted to get a fresh start so that I could see the effect of various XDELTAs and YDELTAs, etc. If the space key was the one pressed, the program would restart. If you plan to

make a competitive game out of BOUNCE, you should substitute other functions for this sequence of instructions.

For example, you could have a timing routine which limited your play to a certain number of seconds. You could decrement a counter each pass through this loop, and stop play when the counter timed out. The timer would have to have several bits of significance, probably at least 24 or 32. An 8-bit timer would time out before you got a chance to even ONCE hit the ball with the paddle!

MUVBAL Routine (1400)

The MUVBAL routine is certainly one of the most important of all. This routine handles all ball movement: it first calculates the new X & Y values and then checks to see if the new position will result in collision with any of the walls, blocks, or paddle.

Ball movement is done a minimum number of pixels at a time, so the ball will appear to move relatively smoothly. Let us take an example to help us. Assume that XDELTA is 2 and YDELTA is also 2. In this case we would want to move the ball 1 pixel in the X direction and 1 pixel in the Y direction. Then we would move the ball the other pixel in both X & Y directions. This completes the move, and a RET is made from MUVBAL.

What if XDELTA and YDELTA are not equal? In this case, we will move the ball like we did above, until either X or Y movement is exhausted, then complete the move changing only the appropriate count. If XDELTA is 1 and YDELTA is 3, the first part of the move is 1 pixel in X direction and 1 pixel in Y direction, as above. The X movement is now exhausted, so the next two moves will each be one pixel in the Y direction only. Details will be discussed in the appropriate subroutines below. After each movement of 1 pixel, checks are made concerning walls, blocks, and paddle. As you can see, the ball will zigzag any time the two are not equal. This is a consequence of low graphics resolution on the TRS-80, and there is nothing we can do about it.

LOITER Routine (1570)

This is a very simple routine, but most important. All the routine does is waste a few milliseconds of time, then calls CHKPDL to see if the left or right arrow keys are pressed. It executes this loop twenty times before returning. The important concept here is that, although we must waste time so that the human can track the path of the ball, we can do something worthwhile by checking to see if paddle movement is called for. If so, we do so in CHKPDL.

So, ball movement is not slowed down by paddle movement! This is something you must accomplish in every similar game you program; no one wants to see the ball almost stop just because the paddle is moving! The number of times the loop is executed (20 in this case) should be varied to allow rapid paddle movement relative to the ball, without moving the paddle so fast that the novice bangs it against one wall and then the other with no fine control. The choice is yours.

VLINE Routine (1670)

There is nothing special about the VLINE routine. It calculates an address by adding 64 (40H), which is the length of a screen line, to the contents of HL and placing the contents of the accumulator at this address. This is done repetitively until the count in B is exhausted. Thus a vertical line is drawn.

SETXY and RSOLXY Routines (1740,1810)

Within the BOUNCE program there are assembly language versions of the SET, RESET, and POINT functions you are probably familiar with from your BASIC programming. Although detailed discussion of these routines will be deferred until we reach them in this listing, let me say now that communication with the routines is done by loading XTEMP with the desired X value and YTEMP with the corresponding Y value. In SETXY we pick up the current values in X and Y, depositing them in XTEMP and YTEMP, respectively. We then call the SET routine.

RSOLXY, beginning in line 1810, resets the old X and Y positions in the same manner.

DIVIDE Routine (1940)

I have previously mentioned that the shift and rotate instructions can be used for multiplication or integer division by a power of two. A left shift by one bit multiplies by two; a right shift by one bit results in integer division by two. For example, if 3 resides in the accumulator:

* Execution of an RLCA instruction would result in a value of 6 in the accumulator.

* Execution of an RRCA instruction would result in a value of 1 in the accumulator.

What if we want neither to multiply by a power of two nor divide by a power of two? We must use something other than simply a series of shifts or rotates!

If the multiplier or divisor is a small number, one commonly uses multiple additions or multiple subtractions. DIVIDE illustrates this technique for division. Upon entry to DIVIDE, the accumulator must contain the dividend, B must contain a zero, and C must contain the divisor. First, the value in register C is subtracted from the value in the accumulator. Then a test is made to see if the resulting number (now in the accumulator) is a negative one. If not, we increment register B and subtract once more. This keeps up until a negative number is obtained, which means we have oversubtracted. We correct this in line 1980 by adding the contents of register C to that in the accumulator.

Upon exit from DIVIDE we have the integer portion of the quotient in register B and the remainder in the accumulator. I'm sure you see a distinct advantage of this routine over the simple rotate described a couple of paragraphs ago -- when we divide 3 by 2 using the DIVIDE routine, we get 1 in register B and 1 in the accumulator. With a rotate or shift we lose the remainder. We will make good use of this remainder, as you will see later.

NEWXY Routine (2030)

This routine calculates the next X and Y positions. This is done starting in line 2030 by picking up the current value of X and storing it away in XOLD. It next checks to see whether further movement in the X direction is warranted. (XCOUNT is decremented each time. A zero value means that the move has been accomplished.) If movement is called for, XCOUNT is decremented and stored away. Then, in lines 2110-2150, a new value for X is calculated by adding the value for XDIRXN to the current value of X.

Let me digress here for a moment to say a little bit about control of ball movement. As you will see later, both XDIRXN and YDIRXN can have values of only +1 or -1. These are the directions the ball can travel. If XDIRXN is +1, the ball is moving to the right; if -1, movement is to the left. If YDIRXN is +1, movement of the ball is from top to bottom of the screen; if -1, from bottom to top.

Perhaps when you first look at the BOUNCE program in operation you will think that there must be a rather complex calculation involved to get the new direction resulting from colliding with a wall, a block, or the paddle. Not so!! All that is required is to find out in which axis the collision occurred (X or Y), and change that direction by reversing the sign of XDIRXN or YDIRXN as appropriate. Try it -- you'll like it! You will see implementation of this below.

Now back to the discussion of NEWXY. You can see that, if XDIRXN is +1, the new X position will be one greater than the old X, XOLD. Conversely, if XDIRXN is -1, the new value will be one less than the old. The same sort of operation is performed for Y and the routine has done its job. All that is left is to POP BC in line 2280 and return.

GTROCL Routine (2330)

Although it is by no means apparent by the mnemonic GTROCL (I only have 6 characters to work with!! What do you expect?), this routine calculates the screen row and column parameters from X and Y values supplied to it by the calling program through XTEMP and YTEMP. In lines 2340-2360 we divide the X value by 2 to get the column. We then store the remainder from the division in MODCOL, to be used later. The integer portion of the quotient is stored in COLUMN. In lines 2400-2450 the same operations are performed to obtain the ROW and MODROW information from Y data.

HTWALL Routine (2540)

After we calculate the new X and Y routines using the NEWXY routine, we want to know if this new position will result in a collision of any sort. The HTWALL routine does this checking as far as the three walls are concerned. If a wall was hit, the direction of movement in the pertinent axis is changed. Collision with any wall will abort the move, since any further movement in the same direction would penetrate the wall, which is a NO-NO! Let us see the details.

First, in lines 2540-2580, we check to see if we are at the left wall. If so we jump to XREV to change XDIRXN. Lines 2590-2650 do the same thing for the right wall. Note that the XREV routine starting in line 2620 using that neat instruction "NEG" to effect the change of sign. Observe that we do not need to know the sign of the number to begin with -- a NEG of any 8-bit value is just like the BASIC statement "A1 = -A1". If the original number was positive, it will now be the same value, but negative, etc.

In lines 2670-2740 we check the top wall in like manner. If any one of the walls was hit, the ZEROCT routine starting in line 2750 zeroes XCOUNT and YCOUNT, so the move will be aborted.

CHKHIT Routine (2860)

The CHKHIT routine starting in line 2860 does the same sort of thing for collisions with a block or the paddle. If a collision occurred, the direction of Y travel is changed. Why just the Y travel? Both the rows of blocks and the paddle lie within the left and right walls, so any collision with them must result in a change only in the Y direction of travel. The direction of X travel can be changed only by collision with either the left or right walls. If the collision was with one of the blocks, we must check to see which row and add to the score accordingly.

In lines 2870-2980, checks are made for possible collision with the paddle. The first check is to see if the current Y value coincides with the Y value for the paddle; if not, there can be no collision with the paddle! If the Y values do agree, then we have one of two possibilities -- we either hit or missed the ball with the paddle. To find out which of these has occurred, we must check to see if the X location of the ball is in the paddle area. We do that by comparing the ball X value to the left-most paddle position first. That is PADLX. If the ball X value is less than PADLX, we missed. If the ball X position is not less than PADLX, we must then use the paddle width, PDLWID, to see whether we hit or missed. The sequence of code in lines 2900-2980 do this checking. If we hit the paddle, line 2980 directs us to YDIRNU.

If we are in the same graphics row as the paddle and we did not hit the ball with the paddle, you guessed it! We missed it!! We must "erase" the ball from its old position, ask for a keypress, wait for that keypress, and restart with a new ball. Line 2990 takes care of the erasure; lines 3000-3020 display the "new ball" message, and line 3030 watches for a keypress, not returning until it gets one. This is preferable to automatically restarting with a new ball, since the harried gameplayer may need to collect wits before continuing! Note that we do nothing with the value of the key pressed, which is returned in the accumulator, so any ordinary key will do. When we get our signal from the keyboard we then wipe out the message and, in line 3070, jump to NUBALL at line 970. NUBALL is in the main routine, and we are in a subroutine, so a jump instruction will leave garbage on the stack. What now, chief? Now you see the wisdom of the first instruction of NUBALL, which restores the original value to the stack pointer. Instant clean!!

Line 3080 labelled YDIRNU begins code which reverses the direction of Y movement. The only thing I need to point out about this is that, in line 3100, we are jumping to YREV, which is a portion of another subroutine (HTWALL). Any time we do that (which may be quite often, to preserve precious memory) we must take care to keep the stack balanced. In this specific case, we have PUSHed BC in line 2860, so we must POP BC in line 3090 before going to YREV. This type of operation is very error-prone, but I still recommend it after you gain some experience, because of significant memory saving. Just be careful.

Earlier in the CHKHIT routine we checked to see if there was a possible collision of the ball with the paddle; if not, we were vectored to line 3110 to check a possible collision with one of the blocks. Lines 3110-3380 check for this and, if so, increment the score.

Concerning the check for possible collision between ball and block, there are two obvious routes to go:

1. Check just those graphics rows known to contain blocks initially.

2. Check all graphics rows to detect collision with anything anywhere on the screen.

We used an approach similar to option 1 for the paddle. I chose to implement option 2 to check for collision between ball and block by checking each and every new X and Y location to see if the pixel is lighted. This will utilize the POINT function. While this will certainly not operate as fast as checking only those Y locations known to contain blocks when the game began, it is much more versatile and more easily applied to different games which may have "random" placement of objects.

Lines 3110-3160 check whether the pixel at the new X and Y positions is lighted. If the pixel is set, the POINT routine will return with the zero flag set; the zero flag will be reset if the pixel is not lighted or set.

Beginning in line 3170 we update the score, since we know a block was struck by the ball (otherwise line 3160 would have vectored us out of the routine). We check to see which row the block was in and increment the score accordingly. In lines 3330-3360 we balance the stack and jump to NUYDIR at line 2720 to reverse the Y direction and reset the pixel so that the block disappears.

GETADR Routine (3430)

This routine, in lines 3430-3580, takes the row and column data calculated by GTROCL and converts these to the corresponding byte address in screen memory. There is nothing new here which needs comment.

NEWPOS and GTPOSN Routines (3620,3660)

The NEWPOS routine in lines 3620-3640 moves the ball on the screen. The GTPOSN routine in lines 3660-3690 takes X and Y data, calculates successively the row and column, the byte address in the screen memory, and the actual graphics character to be stored at that address. Since both routines are nothing more than calls to other routines, there is no need for comment here.

SET Routine (3740)

Discussion of this routine is almost anticlimactic, since we have already discussed most of the routines which actually do the work! GTPOSN returns with the HL register pair containing the address of the specific byte of screen memory and the accumulator containing the value of the graphics character which will set the appropriate pixel on the screen. All that SET has to do is to OR the value in the accumulator with the character already present at the screen memory location. Remember that each byte of screen memory is comprised of 6 pixels; the OR will set the particular pixel of interest without changing any of the others. We must then store the updated byte of data back to the same screen address as shown in line 3760.

Let us run through an example to see how this works. Here are the assumptions:

A contains value of 81H.
HL contains 3C00H.
The byte at 3C00H contains 82H.

```
so:                 1 0 0 0 0 0 0 1      in A
                    1 0 0 0 0 0 1 0      in (HL)

Result after OR:    1 0 0 0 0 0 1 1
```

Let's summarize: The upper left corner of the screen (3C00H) showed a graphics character of 130 (82H, which is just the upper right pixel lighted) before the OR instruction. After the OR, it has a graphics character of 131 (83H), which is where both the top pixels are lighted. You may want to refer back to the table of graphics characters on page 5-7 of this booklet for further study.

RESET Routine (3820)

The RESET routine is in lines 3820-3870 and is similar to the SET routine, but with important differences. A call is made to GTPOSN as in the previous routine. Then we get the inverse of the graphics character which we AND with the contents of the byte of screen memory. Next we OR the new byte with 80H to restore the graphics bias, place the new byte back in the screen address and return.

Let's look at an example of how this works. Here are the assumptions:

A contains 81H.
HL contains 3C00H.
The byte at 3C00H is 83H.

so:	1 0 0 0 0 0 0 1	in A
we first complement it to get the following byte:		
	0 1 1 1 1 1 1 0	in A
	1 0 0 0 0 0 1 1	in (HL)
Result after AND	0 0 0 0 0 0 1 0	in A
Now OR with 80H	1 0 0 0 0 0 0 0	
Result after OR	1 0 0 0 0 0 1 0	in A

Refer again to the table on page 5-7. We start out with both top pixels (83H) of the memory byte set. We wish to reset the upper left pixel (81H). After the above sequence of operations, we end up with 82H, which will display on the TV as just the upper right pixel of the upper left corner of the screen set. How about that?! -- everything works!!

POINT Routine (3930)

By this time I bet that you are one step ahead of me. But just in case, let us look at the POINT routine in lines 3930-3990. Once again we call the GTPOSN routine. This time, we need to save the accumulator in register B temporarily, so we PUSH BC and LD B,A. Next we AND with the present contents of the screen memory address. Finally we compare the value we have after the AND with the original value, now in B. Only if they are identical will the zero flag be set. Just for the fun of it, let us look at one more example.

Take the same assumptions we made in the RESET example. In this case we want to see if the upper left pixel of 3C00H is set.

```
                  1 0 0 0 0 0 0 1    in A
                  1 0 0 0 0 0 1 1    in (HL)
AND               1 0 0 0 0 0 0 1
```

We compare this with the original value, which was 81H; we see that the two values are equal and therefore the zero flag will be set after the AND instruction in line 3960. So everything makes sense at last.

GRAFVL Routine (4030)

We have called this routine several times previously, so you know it is one of the most important ones. Beginning in line 4030, this routine takes the remainder values MODROW and MODCOL we saved way back when and calculates the value of the graphics character. Upon entry to this routine, the HL register pair contains the address of the byte of screen memory, so all we have to do now is find out which one of the 6 pixels is involved. It may help for you to refresh your memory by looking back at page 5-7 one more time at the relationships between the location of a given pixel and its value.

We fetch the value of MODROW into the accumulator. The MODROW value can range from 0-2, since it is the remainder after division by 3. So all we need to do is find out what the value of MODROW is and convert that to a value which will put the pixel into the correct row. If MODROW is 0, the desired pixel is the top one; if 1, the desired pixel is the middle one; if 2, the bottom one. We want to end up with a 1, a 4, or a 10H in the accumulator. We are just considering on which row the pixel of interest is located.

In line 4060 we compare value to 1. If it is equal to 1, we want to load 4 into the accumulator; we are vectored from line 4070, since the Z flag is set, and a 4 is loaded into A in line 4110. If the value in A is not 1, we need to know whether it is greater than or less than 1. The only possible values are 0, 1, and 2, so only one test is necessary; that is in line 4080. If the accumulator is less than 1, the carry flag will be set; the test will fail and fall through to line 4090, where the value (which has to be a 0) is incremented, resulting in 1. If the value is greater than 1, it must be 2 and we are vectored from line 4080 to line 4130 where 10H is loaded into the accumulator.

So we have determined which row the pixel is in -- now starting in line 4140 we determine which column the pixel

is in. There are two columns per byte, as we have previously discussed, and therefore there are only two possible values of MODCOL, 0 and 1. Observation of the aforementioned table on 5-7 will show that we can adjust the value for the right-most column by multiplying the row data by 2. How do we know that the pixel should be in the right-most column? If the value of MODCOL is 0, it is the left column; if 1, the right column. So all we have to do is find out the value of bit 0 of MODCOL and go accordingly. Line 4140 tests bit 0 of MODCOL -- if it is reset (that is, 0), no adjustment of the calculated row data is necessary. Line 4150 vectors us to line 4170 where the graphics bias is added, and we are finished.

If bit 0 of MODCOL is set (that is, a value of 1), the shift instruction in line 4160 will multiply the row data by 2. The graphics bias is added as above, and that's it.

Why did I use "SLA A" which is a two-byte instruction rather than RLCA which is a one-byte instruction? Refer to page B-8 in Appendix B to see what happens when the various shifts are rotates are executed. As I said many times in the audio lectures, we cannot program successfully in assembly language without paying close attention to the flags. How does that apply here? Note that, depending upon whether the value of MODROW was 0 or not, the carry flag will be set or reset. All of the load instructions and the BIT instruction do not affect the carry flag, so we cannot be sure that the carry flag is reset. If we used the RLCA instruction, whatever was in the C flag would be rotated into bit 0. Thus, if the carry flag was set (which it would be if the value of MODROW had been 0) we would have a 1 in the accumulator. Upon execution of the RLCA instruction the accumulator would have a value of 3, rather than the desired value of 2! So we must spend the extra byte of memory and extra 4 clock cycles. Note that the SLA instruction shifts a 0 into bit 0 regardless of the state of the carry flag. As you can see, I'm sure, careful study of Appendix B will be necessary to choose the optimum instructions for the particular situation -- sorry about that!

MAKEUP and WASTE Routines (4210,4240)

Both of these routines are a waste of time! Perhaps I should say that both of the routines are designed to spend a certain amount of time. They are useful, after all! MAKEUP starts in line 4210 and does nothing more than preserve BC, load BC with 60H, and invoke the ROM time delay routine located at 60H. More will be said about the MAKEUP routine and the significance of the 60H in the discussion of the CHKPDL routine below.

The WASTE routine also preserves BC, but rather than loading BC with a constant value set up at assembly time, it picks up the SPEED value we previously calculated. So the amount of time spent by the ROM routine will depend on how the player answered the "skill message" in the main routine. WASTE's mission in life is only to slow down the action to allow the human to react to that action. If faster action is desired, devise a calculation of SPEED which will result in lower values of SPEED -- this will cause the amount of time spent in the ROM routine to be less. Conversely, larger values of SPEED will slow down the action.

CLS Routine (4300)

This routine in lines 4300-4390 clears the screen, but it places "graphics spaces" rather than ordinary spaces in the screen memory like the ROM CLS routine at 1C9H does. What I call a "graphics space" is an 80H, whereas an ordinary space is 20H. They both will show up as a blank on the screen, but the graphics space has bit 7 set, so it already has the graphics bias I have mentioned several times. This is needed in the SET, RESET, and POINT routines.

CHKPDL Routine (4410)

This routine, in lines 4410-4690, checks for the depression of the left and right arrow keys. If either are pressed, it moves the paddle accordingly. Before we discuss how the routine accomplishes this, consider what happens if neither key is pressed. Line 4410 fetches the value from the keyboard as discussed in the section on the keyboard beginning on page 5-5 of this booklet. Line 4420 compares the value obtained with that for the left arrow. If it does not check, line 4430 vectors us to line 4540, where a comparison with the value for the right arrow key is made. If that also does not check, we are vectored to the MAKEUP routine. You will remember the MAKEUP routine as a time waster.

Why do we need a time waster here? Let me digress once again for a short while to discuss this.

Although it is absurdly apparent, I simply must state that "all routines take time"! In this case, time will be taken by the CHKPDL routine if either the left or right arrow key is pressed. Since we want the ball to move the same speed whether or not the paddle is being moved, we must make up for the time which would be spent moving the paddle, by using MAKEUP. The value of 60H as a time delay parameter in the MAKEUP routine (line 4220) was empirically determined, but if you are a purist you can calculate

exactly what the value should be by adding up the clock cycles for all of the pertinent instructions.

The significant point here is that the time should be balanced, so that it does not matter what branching the program does. Earlier I said that, to speed up the action, you should experiment with the SPEED calculations to get smaller or larger values as desired. THAT IS NOT TRUE HERE -- this is simply to make up for the time that the routine would take if one of the arrow keys was pressed. You would not change the amount of time for the MAKEUP routine spends unless you changed the length of time the other branch of the program takes. I hope I have explained this satisfactorily, since it is a very important concept which you must implement for most, if not all, user-interactive programs.

Now let us consider what happens if the left arrow key is pressed. In line 4440 we want to see if the paddle is at the left wall already. If so, we should not take any action except for the MAKEUP routine. If the paddle is not at the left wall, we must move it toward the left. Lines 4470-4500 take care of this; lines 4510-4530 erase the old paddle position.

What about if the right arrow key was pressed? Lines 4560-4640 perform the same sort of operations that we did for moving the paddle to the left. Lines 4650-4690 reset the old paddle position. Actually, the PDLRST segment only resets the one pixel which is no longer needed -- there is no need to completely erase the paddle and then redraw it one pixel over on the screen. It is sufficient just to reset the pixel on the opposite end from the direction of travel and then set one more pixel on the other end!

PDLSET Routine (4710)

Little more has to be said about this routine. It merely sets the pixel at the edge of the existing paddle which causes the appearance of movement in the desired direction.

DRBLOK and **HLINE** Routines (4790,4890)

The DRBLOK routine in lines 4790-4870 draws three rows of blocks on the screen. For each row of blocks the accumulator is loaded with the Y position for the row. The HLINE routine in lines 4890-5000 then draws the row. Both routines should be easy for you to understand at this point in your education.

One comment about technique -- look at lines 4960-4980. What I am doing there is incrementing the XTEMP value (HL is pointed to XTEMP in line 4910) and then comparing it to XHIGH+1. By incrementing the XTEMP value in this manner rather than bringing it into A, incrementing it, and storing it back in XTEMP, we save one byte. Not much, but keep such things in mind because sometimes they will be sorely needed. Assembly language programmers should be continually aware of the expense of instructions both as to their time requirements and memory requirements.

TVMSG Routine (5050)

The TVMSG routine in lines 5050-5160 displays a message whose first character is pointed to by the HL register pair, starting at the screen location pointed to by the DE register pair. For this routine, the message must be followed by a zero byte to allow graceful termination. Otherwise the TVMSG routine just keeps going until it encounters a zero or you press the reset key! The routine is representative of the kind of garden variety code that is not spectacular but is needed in almost every programming task you undertake.

TVSCOR and **TVBYTE** Routines (5210,5330)

As I infer in lines 5180-5190, you will not find many games with a hexadecimal scorekeeper! Surely any commercial program would display the score in good old decimal! I agree, but I did not want the tutorial attributes of the BOUNCE program to be complicated unnecessarily by a binary-to-ASCII-decimal routine. One of the expressions commonly used by my university professors was "This is left as an exercise for the student". I did not like it then, and I feel you are not likely to favor it now, but in this case it is justified!

You are likely to need a binary-to-ASCII-hexadecimal routine like that imbedded in TVSCOR and TVBYTE fairly often as well, so it deserves some study on your part. There is nothing difficult or tricky about the code.

Messages and Variable Storage Area (5510,5620)

I habitually place all messages and storage areas for variables at the end of all of the routines. No particular reason except that they are less likely to interfere with the code if they are back there. If you imbed them in the code you must take pains to jump over them, <u>etc</u>. Of course, if you place them between subroutines there will be no problem. It's totally up to you -- after all, you are a <u>bona fide</u> assembly language programmer by now!

SUGGESTIONS FOR ENHANCEMENT OF BOUNCE

There are several areas where BOUNCE could stand considerable improvement. One obvious one is that you should install a score display routine which shows the score as a decimal number, as discussed above.

You may also want to show the previous high score, to raise the air of competition. How about automatically changing the skill level based on the performance of the player? This could be done by incrementing or decrementing the SPEED variable as indicated after a certain number of hits (or misses, as in my case).

Perhaps the scoring should reflect the skill level; one possibility is to take the appropriate score value and add some arbitrary value to it for each skill level. An easy way would to subtract the original skill level from 10, place that amount in register B, and add the arbitrary value to the score value for the particular hit repetitively till register B is decremented to zero. Thus lower numerical skill values (representing higher degrees of skill) would result in higher scoring for a given number of hits.

One enhancement which I think is really needed is to provide some means of changing the direction of ball movement, i. e., putting "spin" on the ball. Perhaps one should change either the XDELTA or YDELTA variable (or even both of them) if the ball strikes one of the edges of the paddle; no change should result if the ball hits the middle of the paddle. This would add another element of skill to the game. Experimentation will be necessary to see how much and in what direction the change should be.

Let me hear what you come up with in the way of enhancements! That is the kind of thing you must do to become adept at assembly language. As I said at the start, you must practice, practice, practice. Here's wishing you lots of fun!!

```
                    00010 ; 'BOUNCE/SRC' ROUTINE, 11/29/81
                    00020 ;
                    00030 ; THIS PROGRAM IS COPYRIGHTED 1982
                    00040 ; BY JOSEPH E. WILLIS
                    00050 ;    REMSOFT, INC.
                    00060 ;    571 EAST 185
                    00070 ;    EUCLID, OHIO 44119
                    00080 ;
5200                00090          ORG     5200H           ;5600H for Mod 3
                    00100 ;
002B                00110 KEYCHK   EQU     2BH
0049                00120 KEYIN    EQU     49H
0060                00130 DELAY    EQU     60H
3C00                00140 SCREEN   EQU     3C00H
3840                00150 LRKEYS   EQU     3840H
0020                00160 SPACE    EQU     20H
0020                00170 LEFARO   EQU     32
0040                00180 RITARO   EQU     64
0001                00190 ROWMIN   EQU     1
000D                00200 ROWMAX   EQU     13
0000                00210 COLMIN   EQU     0
003F                00220 COLMAX   EQU     63
0004                00230 YLOW     EQU     ROWMIN<1+ROWMIN+1
0028                00240 YHIGH    EQU     ROWMAX<1+ROWMAX+1
0001                00250 XLOW     EQU     COLMIN<1+1
007E                00260 XHIGH    EQU     COLMAX<1
000A                00270 SCORE0   EQU     10
0014                00280 SCORE1   EQU     SCORE0<1
0028                00290 SCORE2   EQU     SCORE0<2
0008                00300 YBLOK2   EQU     YLOW+4
000C                00310 YBLOK1   EQU     YBLOK2+4
0010                00320 YBLOK0   EQU     YBLOK1+4
0010                00330 PDLWID   EQU     16
0029                00340 PADLY    EQU     YHIGH+1
0008                00350 PDLHLF   EQU     PDLWID<-1
3C18                00360 SMPOSN   EQU     SCREEN+18H
3C20                00370 SCPOSN   EQU     SMPOSN+8
3FC0                00380 NBPOSN   EQU     SCREEN+3C0H
                    00390 ;
5200 310D59         00400 BOUNCE   LD      SP,SPEED+300H   ;GIVE PLENTY FOR STAK
5203 CDC854         00410          CALL    CLS             ;CLEAR SCREEN
5206 210000         00420          LD      HL,0            ;CLEAR SCORE
5209 220B56         00430          LD      (SCORE),HL
520C 11183C         00440          LD      DE,SMPOSN       ;SCORE POSN
520F 217E55         00450          LD      HL,SCRMSG       ;SCORE MSG
5212 CD4755         00460          CALL    TVMSG           ;SEND MSG TO TV
5215 3E01           00470          LD      A,ROWMIN        ;DRAW TOP LINE
5217 320356         00480          LD      (ROW),A
521A 3E00           00490          LD      A,COLMIN
521C 320556         00500          LD      (COLUMN),A
521F CD5054         00510          CALL    GETADR
```

```
5222 3E97     00520          LD      A,97H
5224 77       00530          LD      (HL),A
5225 23       00540          INC     HL
5226 3E83     00550          LD      A,83H
5228 063E     00560          LD      B,COLMAX-COLMIN-1
522A 77       00570 B1       LD      (HL),A
522B 23       00580          INC     HL
522C 10FC     00590          DJNZ    B1
522E 3EAB     00600          LD      A,0ABH
5230 77       00610          LD      (HL),A
5231 3E02     00620          LD      A,ROWMIN+1      ;DRAW LEFT VERT LINE
5233 320356   00630          LD      (ROW),A
5236 CD5054   00640          CALL    GETADR
5239 3E95     00650          LD      A,95H
523B CD1153   00660          CALL    VLINE
523E 3E3F     00670          LD      A,COLMAX        ;DRAW RIGHT VERT LINE
5240 320556   00680          LD      (COLUMN),A
5243 CD5054   00690          CALL    GETADR
5246 3EAA     00700          LD      A,0AAH
5248 CD1153   00710          CALL    VLINE
524B 3A0756   00720          LD      A,(PADLX)       ;DRAW PADDLE
524E 0610     00730          LD      B,PDLWID
5250 F5       00740 PDLLUP   PUSH    AF
5251 CD1455   00750          CALL    PDLSET
5254 F1       00760          POP     AF
5255 3C       00770          INC     A
5256 10F8     00780          DJNZ    PDLLUP
5258 CD2055   00790          CALL    DRBLOK          ;DRAW BLOCKS
525B 21CE55   00800 SKILL    LD      HL,LVLMSG       ;GET DESIRED
525E 11C03F   00810          LD      DE,NBPOSN       ;SKILL LEVEL
5261 CD4755   00820          CALL    TVMSG
5264 CD4900   00830          CALL    KEYIN
5267 FE31     00840          CP      '1'             ;REJECT ANYTHING
5269 38F0     00850          JR      C,SKILL         ;NOT BETWEEN
526B FE3A     00860          CP      '9'+1           ;1 AND 9
526D 30EC     00870          JR      NC,SKILL
526F E60F     00880          AND     0FH
5271 07       00890          RLCA                    ;MULTIPLY BY 16
5272 07       00900          RLCA
5273 07       00910          RLCA
5274 07       00920          RLCA
5275 320D56   00930          LD      (SPEED),A
5278 11C03F   00940          LD      DE,NBPOSN       ;ERASE PREV MSG
527B 21A455   00950          LD      HL,BLNKS
527E CD4755   00960          CALL    TVMSG
5281 310D59   00970 NUBALL   LD      SP,SPEED+300H   ;RESET STACK POINTER
5284 ED5F     00980          LD      A,R             ;RANDOM NUM FOR BALL
5286 E67F     00990          AND     7FH             ;POSN BETWEEN 0-127
5288 FE01     01000          CP      XLOW
528A 3002     01010          JR      NC,B2
528C 3E01     01020          LD      A,XLOW
528E FE7E     01030 B2       CP      XHIGH
```

```
5290 3802      01040          JR      C,B3
5292 3E7E      01050          LD      A,XHIGH          ;IF >=XHIGH, MAKE
XHIGH
5294 32F755    01060 B3       LD      (X),A
5297 3E11      01070          LD      A,YBLOK0+1       ;ROW JUST UNDER
BLOCKS
5299 32F855    01080          LD      (Y),A
529C ED5F      01090          LD      A,R              ;RAND NUM FOR XDELTA
529E F5        01100          PUSH    AF               ;& XDIRXN -- SAVE
529F E603      01110          AND     3
52A1 2002      01120          JR      NZ,B4
52A3 3E01      01130          LD      A,1
52A5 32FE55    01140 B4       LD      (XDELTA),A
52A8 F1        01150          POP     AF
52A9 FE3F      01160          CP      3FH
52AB 3EFF      01170          LD      A,0FFH           ; -1
52AD 3002      01180          JR      NC,B4A
52AF 3C        01190          INC     A
52B0 3C        01200          INC     A                ; +1
52B1 32FD55    01210 B4A      LD      (XDIRXN),A
52B4 ED5F      01220          LD      A,R              ;RAND NUM FOR YDELTA
52B6 E603      01230          AND     3
52B8 2002      01240          JR      NZ,B5
52BA 3E01      01250          LD      A,1
52BC 320156    01260 B5       LD      (YDELTA),A
52BF 3E01      01270          LD      A,1              ; +1
52C1 320056    01280 B5A      LD      (YDIRXN),A
52C4 CD1B53    01290          CALL    SETXY            ;SHOW BALL
52C7 CDDD52    01300 PONGLP   CALL    MUVBAL           ;MOVE BALL
52CA CD0253    01310          CALL    LOITER           ;WASTE TIME
52CD CD5555    01320          CALL    TVSCOR           ;UPDATE SCORE
52D0 CD2B00    01330          CALL    KEYCHK           ;KEY PRESSED?
52D3 B7        01340          OR      A
52D4 28F1      01350          JR      Z,PONGLP         ;NO, KEEP GOING
52D6 FE20      01360          CP      SPACE            ;YES, SPACE KEY?
52D8 20ED      01370          JR      NZ,PONGLP        ;NO
52DA C30052    01380          JP      BOUNCE           ;YES, START OVER
               01390 ;
52DD C5        01400 MUVBAL   PUSH    BC
52DE 3AFE55    01410          LD      A,(XDELTA)
52E1 32FF55    01420          LD      (XCOUNT),A
52E4 3A0156    01430          LD      A,(YDELTA)
52E7 320256    01440          LD      (YCOUNT),A
52EA CD4453    01450 MUVLUP   CALL    NEWXY            ;CALC NEW X & Y
52ED CDA053    01460          CALL    HTWALL           ;HIT A WALL?
52F0 CDDB53    01470          CALL    CHKHIT           ;HIT BLOCK OR PADDLE?
52F3 CD6A54    01480          CALL    NEWPOS           ;SHOW NEW POSITION
52F6 3AFF55    01490          LD      A,(XCOUNT)       ;FINISHED WITH MOVE?
52F9 47        01500          LD      B,A
52FA 3A0256    01510          LD      A,(YCOUNT)
52FD B0        01520          OR      B
52FE 20EA      01530          JR      NZ,MUVLUP        ;NO
```

```
5300 C1        01540          POP     BC              ;YES
5301 C9        01550          RET
               01560 ;
5302 C5        01570 LOITER   PUSH    BC
5303 0614      01580          LD      B,20
5305 C5        01590 LOITLP   PUSH    BC
5306 CDD754    01600          CALL    CHKPDL          ;NEED TO MOVE PADDLE?
5309 CDBE54    01610          CALL    WASTE           ;WASTE TIME
530C C1        01620          POP     BC
530D 10F6      01630          DJNZ    LOITLP
530F C1        01640          POP     BC
5310 C9        01650          RET
               01660 ;
5311 114000    01670 VLINE    LD      DE,40H
5314 060C      01680          LD      B,ROWMAX-ROWMIN
5316 77        01690 L1       LD      (HL),A
5317 19        01700          ADD     HL,DE
5318 10FC      01710          DJNZ    L1
531A C9        01720          RET
               01730 ;
531B 3AF755    01740 SETXY    LD      A,(X)
531E 32FB55    01750          LD      (XTEMP),A
5321 3AF855    01760          LD      A,(Y)
5324 32FC55    01770          LD      (YTEMP),A
5327 CD7B54    01780          CALL    SET
532A C9        01790          RET
               01800 ;
532B 3AF955    01810 RSOLXY   LD      A,(XOLD)
532E 32FB55    01820          LD      (XTEMP),A
5331 3AFA55    01830          LD      A,(YOLD)
5334 32FC55    01840          LD      (YTEMP),A
5337 CD8154    01850          CALL    RESET
533A C9        01860          RET
               01870 ;
               01880 ; 8-BIT DIVIDE ROUTINE
               01890 ;  ENTRY: DIVIDEND IN A
               01900 ;         DIVISOR IN BC
               01910 ;  EXIT:  INTEGER RESULT IN B
               01920 ;         REMAINDER IN A
               01930 ;
533B 91        01940 DIVIDE   SUB     C
533C FA4253    01950          JP      M,DIVEND
533F 04        01960          INC     B
5340 18F9      01970          JR      DIVIDE
5342 81        01980 DIVEND   ADD     A,C
5343 C9        01990          RET
               02000 ;
               02010 ; CALCULATES NEW X & Y POSITIONS
               02020 ;
5344 C5        02030 NEWXY    PUSH    BC
5345 3AF755    02040          LD      A,(X)           ;FETCH X AND
5348 32F955    02050          LD      (XOLD),A        ;ASSIGN TO OLD X
```

```
534B 3AFF55    02060          LD      A,(XCOUNT)      ;DONE WITH X MOVE?
534E B7        02070          OR      A
534F 280F      02080          JR      Z,NEWY          ;YES
5351 3D        02090          DEC     A               ;NO, UPDATE COUNT
5352 32FF55    02100          LD      (XCOUNT),A
5355 3AF755    02110          LD      A,(X)           ;UPDATE X POSN
5358 47        02120          LD      B,A
5359 3AFD55    02130          LD      A,(XDIRXN)
535C 80        02140          ADD     A,B
535D 32F755    02150          LD      (X),A
5360 3AF855    02160 NEWY     LD      A,(Y)           ;FETCH Y AND ASSIGN
5363 32FA55    02170          LD      (YOLD),A        ;TO OLD Y
5366 3A0256    02180          LD      A,(YCOUNT)      ;DONE WITH Y MOVE?
5369 B7        02190          OR      A
536A 280F      02200          JR      Z,NUYOUT        ;YES, FINISHED
536C 3D        02210          DEC     A               ;NO, UPDATE COUNT
536D 320256    02220          LD      (YCOUNT),A
5370 3AF855    02230          LD      A,(Y)           ;UPDATE Y POSN
5373 47        02240          LD      B,A
5374 3A0056    02250          LD      A,(YDIRXN)
5377 80        02260          ADD     A,B
5378 32F855    02270          LD      (Y),A
537B C1        02280 NUYOUT   POP     BC
537C C9        02290          RET
               02300 ;
               02310 ; GIVEN X & Y POSITIONS, CALCULATES ROW & COLUMN
               02320 ;
537D C5        02330 GTROCL   PUSH    BC
537E 3AFB55    02340          LD      A,(XTEMP)       ;GET COLUMN
5381 010200    02350          LD      BC,2
5384 CD3B53    02360          CALL    DIVIDE
5387 320656    02370          LD      (MODCOL),A
538A 78        02380          LD      A,B
538B 320556    02390          LD      (COLUMN),A
538E 3AFC55    02400          LD      A,(YTEMP)       ;GET ROW
5391 010300    02410          LD      BC,3
5394 CD3B53    02420          CALL    DIVIDE
5397 320456    02430          LD      (MODROW),A
539A 78        02440          LD      A,B
539B 320356    02450          LD      (ROW),A
539E C1        02460          POP     BC
539F C9        02470          RET
               02480 ;
               02490 ; CHECKS WHETHER NEW X & Y POSITIONS WILL CAUSE
               02500 ; A HIT ON TOP, LEFT, OR RIGHT WALLS.
               02510 ;
               02520 ; IF SO, CHANGE DIRECTION OF MOVEMENT
               02530 ;
53A0 3AF755    02540 HTWALL   LD      A,(X)
53A3 FE02      02550          CP      XLOW+1          ;AT LEFT WALL?
53A5 3004      02560          JR      NC,HTW1         ;NO
53A7 3E01      02570          LD      A,XLOW          ;YES, REV DIRXN
```

```
53A9 1806      02580          JR      XREV
53AB FE7E      02590 HTW1     CP      XHIGH           ;AT RIGHT WALL?
53AD 3810      02600          JR      C,HTW2          ;NO
53AF 3E7E      02610          LD      A,XHIGH         ;YES
53B1 32F755    02620 XREV     LD      (X),A           ;SET X AND
53B4 3AFD55    02630          LD      A,(XDIRXN)      ;CHANGE SIGN OF
53B7 ED44      02640          NEG                     ;X DIRECTION
53B9 32FD55    02650          LD      (XDIRXN),A
53BC CDD253    02660          CALL    ZEROCT
53BF 3AF855    02670 HTW2     LD      A,(Y)           ;PAST TOP WALL?
53C2 FE04      02680          CP      YLOW
53C4 D0        02690          RET     NC              ;NO
53C5 3E04      02700          LD      A,YLOW          ;YES, SET Y
53C7 32F855    02710 YREV     LD      (Y),A
53CA 3A0056    02720 NUYDIR   LD      A,(YDIRXN)      ;AND CHANGE SIGN
53CD ED44      02730          NEG                     ;OF Y DIRECTION
53CF 320056    02740          LD      (YDIRXN),A
53D2 3E00      02750 ZEROCT   LD      A,0             ;IF ANY WALL WAS HIT,
53D4 32FF55    02760          LD      (XCOUNT),A      ;ABORT THE MOVE BY
53D7 320256    02770          LD      (YCOUNT),A      ;ZEROING BOTH COUNTS
53DA C9        02780          RET
               02790 ;
               02800 ; CHECKS TO SEE IF NEW X & Y WILL CAUSE THE BALL
               02810 ; TO HIT THE PADDLE OR ANY BLOCK
               02820 ;
               02830 ; IF SO, CHANGE DIRECTION OF TRAVEL
               02840 ; IF BLOCK, ADD TO SCORE.
               02850 ;
53DB C5        02860 CHKHIT   PUSH    BC
53DC 3AF855    02870          LD      A,(Y)
53DF FE28      02880          CP      YHIGH           ;POSSIBLE HIT ON
PADDLE?
53E1 3830      02890          JR      C,HTBLOK        ;NO, CHK HIT ON BLOCK
53E3 3A0756    02900          LD      A,(PADLX)       ;GET X POSN OF PADDLE
53E6 47        02910          LD      B,A             ;STORE IN B
53E7 C610      02920          ADD     A,PDLWID        ;ADD WIDTH OF PADDLE
53E9 4F        02930          LD      C,A             ;SAVE IN C
53EA 3AF755    02940          LD      A,(X)           ;GET X POSN OF BALL
53ED B8        02950          CP      B               ;< LEFT SIDE OF
PADDLE?
53EE 3803      02960          JR      C,NOHIT         ;YES, NO HIT
53F0 B9        02970          CP      C               ;ON PADDLE?
53F1 381B      02980          JR      C,YDIRNU        ;YES
53F3 CD2B53    02990 NOHIT    CALL    RSOLXY          ;NO RESET OLD X & Y
53F6 11C03F    03000          LD      DE,NBPOSN       ;DSPLY "NEW BALL" MSG
53F9 218555    03010          LD      HL,HITMSG
53FC CD4755    03020          CALL    TVMSG
53FF CD4900    03030          CALL    KEYIN           ;WAIT FOR KEY PRESS
5402 11C03F    03040          LD      DE,NBPOSN       ;ERASE MESSAGE
5405 21A455    03050          LD      HL,BLNKS
5408 CD4755    03060          CALL    TVMSG
540B C38152    03070          JP      NUBALL          ;SERVE NEW BALL
```

```
540E 3E28      03080 YDIRNU  LD      A,YHIGH          ;REVERSE Y DIRECTION
5410 C1        03090         POP     BC               ;BALANCE STACK
5411 18B4      03100         JR      YREV
5413 3AF855    03110 HTBLOK  LD      A,(Y)            ;WILL NEW X & Y
5416 32FC55    03120         LD      (YTEMP),A        ;VALUES HIT A BLOCK?
5419 3AF755    03130         LD      A,(X)
541C 32FB55    03140         LD      (XTEMP),A
541F CD8A54    03150         CALL    POINT
5422 202A      03160         JR      NZ,CHKOUT        ;NO HIT
5424 D5        03170         PUSH    DE
5425 E5        03180         PUSH    HL
5426 3AF855    03190         LD      A,(Y)            ;HIT, WHICH ROW?
5429 FE0C      03200         CP      YBLOK1
542B 2005      03210         JR      NZ,HTB1
542D 111400    03220         LD      DE,SCORE1
5430 180C      03230         JR      UPSCOR
5432 FE08      03240 HTB1    CP      YBLOK2
5434 2005      03250         JR      NZ,HTB2
5436 112800    03260         LD      DE,SCORE2
5439 1803      03270         JR      UPSCOR
543B 110A00    03280 HTB2    LD      DE,SCORE0
543E 2A0B56    03290 UPSCOR  LD      HL,(SCORE)       ;UPDATE SCORE
5441 19        03300         ADD     HL,DE
5442 220B56    03310         LD      (SCORE),HL
5445 CD8154    03320         CALL    RESET            ;ERASE THE BLOCK
5448 E1        03330         POP     HL               ;BALANCE STACK
5449 D1        03340         POP     DE
544A C1        03350         POP     BC
544B C3CA53    03360         JP      NUYDIR           ;REVERSE Y DIRECTION
544E C1        03370 CHKOUT  POP     BC
544F C9        03380         RET
               03390 ;
               03400 ; CONVERT ROW & COLUMN DATA TO THE CORRESPONDING
               03410 ; ACTUAL SCREEN MEMORY ADDRESS.
               03420 ;
5450 C5        03430 GETADR  PUSH    BC
5451 D5        03440         PUSH    DE
5452 21003C    03450         LD      HL,SCREEN
5455 114000    03460         LD      DE,40H
5458 3A0356    03470         LD      A,(ROW)
545B A7        03480         AND     A                ;CHK FOR ZERO
545C 2804      03490         JR      Z,GTA2
545E 47        03500         LD      B,A
545F 19        03510 GTA1    ADD     HL,DE
5460 10FD      03520         DJNZ    GTA1
5462 3A0556    03530 GTA2    LD      A,(COLUMN)
5465 5F        03540         LD      E,A
5466 19        03550         ADD     HL,DE
5467 D1        03560         POP     DE
5468 C1        03570         POP     BC
5469 C9        03580         RET
               03590 ;
```

```
                03600 ; MOVES "BALL" ON SCREEN
                03610 ;
546A CD2B53     03620 NEWPOS  CALL    RSOLXY
546D CD1B53     03630         CALL    SETXY
5470 C9         03640         RET
                03650 ;
5471 CD7D53     03660 GTPOSN  CALL    GTROCL          ;CALC ROW & COLUMN
5474 CD5054     03670         CALL    GETADR          ;GET ADDRESS OF R&C
5477 CD9354     03680         CALL    GRAFVL          ;CALC GRAPHICS CHAR
547A C9         03690         RET
                03700 ;
                03710 ; ROUTINE TO SET ANY POINT ON SCREEN
                03720 ; FUNCTIONALLY SAME AS "SET(X,Y)" IN BASIC
                03730 ;
547B CD7154     03740 SET     CALL    GTPOSN
547E B6         03750         OR      (HL)
547F 77         03760         LD      (HL),A
5480 C9         03770         RET
                03780 ;
                03790 ; ROUTINE TO RESET ANY POINT ON SCREEN
                03800 ; FUNCTIONALLY SAME AS "RESET(X,Y)" IN BASIC
                03810 ;
5481 CD7154     03820 RESET   CALL    GTPOSN
5484 2F         03830         CPL                     ;GET INVERSE
5485 A6         03840         AND     (HL)
5486 F680       03850         OR      80H
5488 77         03860         LD      (HL),A
5489 C9         03870         RET
                03880 ;
                03890 ; POINT ROUTINE, FUNCTIONALLY SAME AS "POINT(X,Y)" IN
BASIC
                03900 ; Z FLAG WILL BE: SET IF POINT IS LIGHTED
                03910 ;                 RESET IF POINT IS NOT LIGHTED
                03920 ;
548A CD7154     03930 POINT   CALL    GTPOSN
548D C5         03940         PUSH    BC
548E 47         03950         LD      B,A
548F A6         03960         AND     (HL)
5490 B8         03970         CP      B
5491 C1         03980         POP     BC
5492 C9         03990         RET
                04000 ;
                04010 ; CALCULATES THE VALUE OF THE GRAPHICS CHARACTER
                04020 ;
5493 3A0456     04030 GRAFVL  LD      A,(MODROW)
5496 DDE5       04040         PUSH    IX
5498 DD210656   04050         LD      IX,MODCOL
549C FE01       04060         CP      1
549E 2805       04070         JR      Z,P1
54A0 3007       04080         JR      NC,P2
54A2 3C         04090         INC     A
54A3 1806       04100         JR      ADJPOS
```

```
54A5 3E04      04110 P1       LD       A,4
54A7 1802      04120          JR       ADJPOS
54A9 3E10      04130 P2       LD       A,10H
54AB DDCB0046  04140 ADJPOS   BIT      0,(IX+0)        ;ADJUST FOR COLUMN
54AF 2802      04150          JR       Z,ADJVAL        ;NO ADJ NEEDED IF
ZERO
54B1 CB27      04160          SLA      A               ;MULT BY 2
54B3 F680      04170 ADJVAL   OR       80H
54B5 DDE1      04180          POP      IX
54B7 C9        04190          RET
               04200 ;
54B8 C5        04210 MAKEUP   PUSH     BC
54B9 016000    04220          LD       BC,60H
54BC 1805      04230          JR       W1
54BE C5        04240 WASTE    PUSH     BC
54BF ED4B0D56  04250          LD       BC,(SPEED)
54C3 CD6000    04260 W1       CALL     DELAY
54C6 C1        04270          POP      BC
54C7 C9        04280          RET
               04290 ;
54C8 E5        04300 CLS      PUSH     HL
54C9 21003C    04310          LD       HL,SCREEN
54CC 3E80      04320 CLSLUP   LD       A,80H
54CE 77        04330          LD       (HL),A
54CF 23        04340          INC      HL
54D0 7C        04350          LD       A,H
54D1 FE40      04360          CP       SCREEN+400H<-8
54D3 20F7      04370          JR       NZ,CLSLUP
54D5 E1        04380          POP      HL
54D6 C9        04390          RET
               04400 ;
54D7 3A4038    04410 CHKPDL   LD       A,(LRKEYS)
54DA FE20      04420          CP       LEFARO          ;LEFT ARROW PRESSED?
54DC 2014      04430          JR       NZ,CHKRIT       ;NO, CHK RIGHT ARROW
54DE 3A0756    04440          LD       A,(PADLX)       ;YES, GET PADDLE POSN
54E1 FE01      04450          CP       XLOW            ;AT LEFT WALL?
54E3 28D3      04460          JR       Z,MAKEUP        ;YES, SPEND SOME TIME
54E5 3D        04470          DEC      A               ;NO, MOVE LEFT
54E6 F5        04480 CL1      PUSH     AF              ;SAVE FOR LATER USE
54E7 320756    04490          LD       (PADLX),A       ;UPDATE PADDLE COLUMN
54EA CD1455    04500          CALL     PDLSET          ;SET NEW PIXEL
54ED F1        04510          POP      AF              ;RETRIEVE OLD PIXEL
LOCN
54EE C611      04520          ADD      A,PDLWID+1      ;RESET OLD LOCN
54F0 1816      04530          JR       PDLRST
54F2 FE40      04540 CHKRIT   CP       RITARO          ;RIGHT ARROW PRESSED?
54F4 20C2      04550          JR       NZ,MAKEUP       ;NO, IGNORE KEY
54F6 3A0756    04560          LD       A,(PADLX)
54F9 FE6E      04570          CP       XHIGH-PDLWID    ;AT RIGHT WALL?
54FB 28BB      04580          JR       Z,MAKEUP        ;YES, MOVE NO FURTHER
54FD F5        04590          PUSH     AF              ;SAVE FOR LATER USE
54FE 3C        04600          INC      A
```

```
54FF 320756   04610          LD      (PADLX),A       ;UPDATE PADDLE COLUMN
5502 C610     04620          ADD     A,PDLWID        ;SET NEW PIXEL
5504 CD1455   04630          CALL    PDLSET
5507 F1       04640          POP     AF
5508 32FB55   04650 PDLRST   LD      (XTEMP),A       ;RESET OLD PIXEL
550B 3E29     04660          LD      A,PADLY
550D 32FC55   04670          LD      (YTEMP),A
5510 CD8154   04680          CALL    RESET
5513 C9       04690          RET
              04700 ;
5514 32FB55   04710 PDLSET   LD      (XTEMP),A
5517 3E29     04720          LD      A,PADLY
5519 32FC55   04730          LD      (YTEMP),A
551C CD7B54   04740          CALL    SET
551F C9       04750          RET
              04760 ;
              04770 ; DRAW THREE ROWS OF BLOCKS
              04780 ;
5520 E5       04790 DRBLOK   PUSH    HL
5521 3E0C     04800          LD      A,YBLOK1
5523 CD3255   04810          CALL    HLINE
5526 3E08     04820          LD      A,YBLOK2
5528 CD3255   04830          CALL    HLINE
552B 3E10     04840          LD      A,YBLOK0
552D CD3255   04850          CALL    HLINE
5530 E1       04860          POP     HL
5531 C9       04870          RET
              04880 ;
5532 32FC55   04890 HLINE    LD      (YTEMP),A
5535 3E01     04900          LD      A,XLOW
5537 21FB55   04910          LD      HL,XTEMP
553A 77       04920          LD      (HL),A
553B E5       04930 DRLUP    PUSH    HL
553C CD7B54   04940          CALL    SET
553F E1       04950          POP     HL
5540 34       04960          INC     (HL)
5541 7E       04970          LD      A,(HL)
5542 FE7F     04980          CP      XHIGH+1
5544 38F5     04990          JR      C,DRLUP
5546 C9       05000          RET
              05010 ;
              05020 ; DISPLAYS THE MESSAGE POINTED TO BY HL REGS
              05030 ; AT THE SCREEN ADDRESS POINTED TO BY DE REGS.
              05040 ;
5547 D5       05050 TVMSG    PUSH    DE
5548 E5       05060          PUSH    HL
5549 7E       05070 TVMSG1   LD      A,(HL)
554A B7       05080          OR      A
554B 2805     05090          JR      Z,TVMOUT
554D 12       05100          LD      (DE),A
554E 13       05110          INC     DE
554F 23       05120          INC     HL
```

```
5550 18F7      05130          JR      TVMSG1
5552 E1        05140 TVMOUT   POP     HL
5553 D1        05150          POP     DE
5554 C9        05160          RET
               05170 ;
               05180 ; THIS GAME HAS WHAT IS PROBABLY A UNIQUE FEATURE
               05190 ;    *****  A HEXADECIMAL SCOREKEEPER!!!  ****
               05200 ;
5555 D5        05210 TVSCOR   PUSH    DE
5556 E5        05220          PUSH    HL
5557 11203C    05230          LD      DE,SCPOSN
555A 2A0B56    05240          LD      HL,(SCORE)      ;FETCH SCORE
555D 7C        05250          LD      A,H             ;DISPLAY MOST SIG
BYTE
555E CD6855    05260          CALL    TVBYTE
5561 7D        05270          LD      A,L             ;DISPLAY LSB
5562 CD6855    05280          CALL    TVBYTE
5565 E1        05290          POP     HL
5566 D1        05300          POP     DE
5567 C9        05310          RET
               05320 ;
5568 F5        05330 TVBYTE   PUSH    AF
5569 0F        05340          RRCA                    ;DIVIDE BY 16
556A 0F        05350          RRCA
556B 0F        05360          RRCA
556C 0F        05370          RRCA
556D CD7155    05380          CALL    TVNIBL          ;DISPLAY NIBBLE
5570 F1        05390          POP     AF
5571 E60F      05400 TVNIBL   AND     0FH
5573 F630      05410          OR      '0'             ;ADJUST FOR ASCII
BIAS
5575 FE3A      05420          CP      '9'+1           ;DIGIT?
5577 3802      05430          JR      C,TVOUT         ;YES
5579 C607      05440          ADD     A,7             ;NO, ADD ALPHA BIAS
557B 12        05450 TVOUT    LD      (DE),A
557C 13        05460          INC     DE
557D C9        05470          RET
               05480 ;
               05490 ; MESSAGE AREA
               05500 ;
557E 53        05510 SCRMSG   DEFM    'SCORE:'
557F 43
5580 4F
5581 52
5582 45
5583 3A
5584 00        05520          DEFB    0
5585 50        05530 HITMSG   DEFM    'PRESS ANY KEY FOR ANOTHER BALL'
5586 52
5587 45
5588 53
5589 53
```

```
558A 20
558B 41
558C 4E
558D 59
558E 20
558F 4B
5590 45
5591 59
5592 20
5593 46
5594 4F
5595 52
5596 20
5597 41
5598 4E
5599 4F
559A 54
559B 48
559C 45
559D 52
559E 20
559F 42
55A0 41
55A1 4C
55A2 4C
55A3 00        05540          DEFB    0
55A4 20        05550 BLNKS    DEFM    '
    '
55A5 20
55A6 20
55A7 20
55A8 20
55A9 20
55AA 20
55AB 20
55AC 20
55AD 20
55AE 20
55AF 20
55B0 20
55B1 20
55B2 20
55B3 20
55B4 20
55B5 20
55B6 20
55B7 20
55B8 20
55B9 20
55BA 20
55BB 20
55BC 20
```

```
55BD 20
55BE 20
55BF 20
55C0 20
55C1 20
55C2 20
55C3 20
55C4 20
55C5 20
55C6 20
55C7 20
55C8 20
55C9 20
55CA 20
55CB 20
55CC 20
55CD 00         05560           DEFB    0
55CE 45         05570 LVLMSG    DEFM    'ENTER 1-9 FOR SKILL LEVEL (9 IS
EASIEST)'
55CF 4E
55D0 54
55D1 45
55D2 52
55D3 20
55D4 31
55D5 2D
55D6 39
55D7 20
55D8 46
55D9 4F
55DA 52
55DB 20
55DC 53
55DD 4B
55DE 49
55DF 4C
55E0 4C
55E1 20
55E2 4C
55E3 45
55E4 56
55E5 45
55E6 4C
55E7 20
55E8 28
55E9 39
55EA 20
55EB 49
55EC 53
55ED 20
55EE 45
55EF 41
```

```
55F0 53
55F1 49
55F2 45
55F3 53
55F4 54
55F5 29
55F6 00          05580          DEFB    0
                 05590 ;
                 05600 ; VARIABLE STORAGE AREA
                 05610 ;
0001             05620 X        DEFS    1
0001             05630 Y        DEFS    1
0001             05640 XOLD     DEFS    1
0001             05650 YOLD     DEFS    1
0001             05660 XTEMP    DEFS    1
0001             05670 YTEMP    DEFS    1
0001             05680 XDIRXN   DEFS    1
0001             05690 XDELTA   DEFS    1
0001             05700 XCOUNT   DEFS    1
0001             05710 YDIRXN   DEFS    1
0001             05720 YDELTA   DEFS    1
0001             05730 YCOUNT   DEFS    1
0001             05740 ROW      DEFS    1
0001             05750 MODROW   DEFS    1
0001             05760 COLUMN   DEFS    1
0001             05770 MODCOL   DEFS    1
5607 36          05780 PADLX    DEFB    XHIGH-XLOW<-1-PDLHLF
0001             05790 POSVAL   DEFS    1
0002             05800 ADDR     DEFS    2
0002             05810 SCORE    DEFS    2
560D 0000        05820 SPEED    DEFW    0
5200             05830          END     BOUNCE
00000 Total Errors
```

Appendix A
Menus for Audio Lessons

For your convenience, here are the menus for the display programs. These are the same as you will see when you load a particular display module and depress the <M> key. Although these are provided, I strongly suggest that you begin with the first lesson and continue in numerical order. This listing should be most helpful to you in your review after one pass through the entire course.

SESSIONS 1 and 2

A) Binary Numbering System
B) Octal Representation
C) Hexadecimal Representation
D) Binary Addition
E) Two's Complement
F) Arithmetic Flags
G) Boolean Operators
H) Flags and the Boolean Operators
I) *** SESSION 2 ***
J) Available Resources
K) Z80 Registers
L) Assembly Language Format

SESSIONS 3 and 4

A) 8-Bit Load Instructions
B) Overview of 8-Bit Load Group
C) 16-Bit Load Instructions
D) Overview of 16-Bit Load Group
E) *** SESSION 4 ***
F) Interrupts
G) 8-Bit Arithmetic & Logical Group
H) 16-Bit Arithmetic Group

SESSIONS 5 and 6

A) Jumps
B) Calls and Returns
C) Rotate & Shift Group
D) Bit Set, Reset, and Test Group
E) Input and Output Group
F) Exchange Group
G) Block Transfer & Search Group

SESSIONS 5 and 6, Continued

H) *** SESSION 6 ***
I) Assembler Pseudo Opcodes
J) QDMTST Overview
K) Equates and Data Storage Handling
L) QDMTST Initialization Section & Test Loop

SESSIONS 7 and 8

A) QDMTST Error Handling Routine
B) TV Routine
C) Miscellaneous Display Routines
D) Automatic Handling of Starting Address for Testing
E) *** SESSION 8 ***
F) KBTVMOD Overview
G) KBDTVMOD Equates and Data Storage Requirements
H) KBDTVMOD Set-up Routine
I) Routine to get Printer Parameters
J) Keyboard Modification Routine

SESSIONS 9 and 10

A) Video Modification Routine
B) Printer Routines
C) Message Output Routine
D) GETNUM Routine
E) FORMS Routine
F) Bottom-of-Page Routine
G) *** SESSION 10 ***
H) Software Development
I) Principles of Program Design
J) Approaches to Program Design
K) Common Errors in Assembly Language Programming
L) Debugging Considerations

Appendix B

SUMMARY OF Z80 INSTRUCTIONS AND EXECUTION TIMES

Reprinted from the
ZILOG Z80 CPU TECHNICAL MANUAL

Instruction	C	Z	P/V	S	N	H	Comments
ADD s; ADC s	‡	‡	V	‡	0	‡	8-bit add or add with carry
SUB s; SBC s, CP s, NEG	‡	‡	V	‡	1	‡	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	0	‡	P	‡	0	1	Logical operations
OR s; XOR x	0	‡	P	‡	0	0	And set's different flags
INC s	•	‡	V	‡	0	‡	8-bit increment
DEC s	•	‡	V	‡	1	‡	8-bit decrement
ADD DD, SS	‡	•	•	•	0	X	16-bit add
ADC HL, SS	‡	‡	V	‡	0	X	16-bit add with carry
SBC HL, SS	‡	‡	V	‡	1	X	16-bit subtract with carry
RLA; RLCA, RRA, RRCA	‡	•	•	•	0	0	Rotate accumulator
RL s; RLC s; RR s; RRC s SLA s; SRA s; SRL s	‡	‡	P	‡	0	0	Rotate and shift location s
RLD, RRD	•	‡	P	‡	0	0	Rotate digit left and right
DAA	‡	‡	P	‡	•	‡	Decimal adjust accumulator
CPL	•	•	•	•	1	1	Complement accumulator
SCF	1	•	•	•	0	0	Set carry
CCF	‡	•	•	•	0	X	Complement carry
IN r, (C)	•	‡	P	‡	0	0	Input register indirect
INI; IND; OUTI; OUTD	•	‡	X	X	1	X	Block input and output
INIR; INDR; OTIR; OTDR	•	1	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD	•	X	‡	X	0	0	Block transfer instructions
LDIR, LDDR	•	X	0	X	0	0	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	•	‡	‡	X	1	X	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	•	‡	IFF	‡	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	•	‡	X	X	0	1	The state of bit b of location s is copied into the Z flag

The following notation is used in this table:

Symbol	Operation
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from into bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
‡	The flag is affected according to the result of the operation.
•	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction. .
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>

8-bit loads

No. flags are affected!

clock cycles, each about 1/2 μs

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	OP-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T Cycles	Comments r, r'	Reg.
LD r, r'	r ← r'	•	•	•	•	•	•	01 r r'	1	1	4	000	B
LD r, n	r ← n	•	•	•	•	•	•	00 r 110	2	2	7	001	C
								← n →				010	D
LD r, (HL)	r ← (HL)	•	•	•	•	•	•	01 r 110	1	2	7	011	E
LD r, (IX+d)	r ← (IX+d)	•	•	•	•	•	•	11 011 101	3	5	19	100	H
								01 r 110				101	L
								← d →					
LD r, (IY+d)	r ← (IY+d)	•	•	•	•	•	•	11 111 101	3	5	19	111	A
								01 r 110					
								← d →					
LD (HL), r	(HL) ← r	•	•	•	•	•	•	01 110 r	1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	•	•	•	•	11 011 101	3	5	19		
								01 110 r					
								← d →					
LD (IY+d), r	(IY+d) ← r	•	•	•	•	•	•	11 111 101	3	5	19		
								01 110 r					
								← d →					
LD (HL), n	(HL) ← n	•	•	•	•	•	•	00 110 110	2	3	10		
								← n →					
LD (IX+d), n	(IX+d) ← n	•	•	•	•	•	•	11 011 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD (IY+d), n	(IY+d) ← n	•	•	•	•	•	•	11 111 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD A, (BC)	A ← (BC)	•	•	•	•	•	•	00 001 010	1	2	7		
LD A, (DE)	A ← (DE)	•	•	•	•	•	•	00 011 010	1	2	7		
LD A, (nn)	A ← (nn)	•	•	•	•	•	•	00 111 010	3	4	13		
								← n →					
								← n →					
LD (BC), A	(BC) ← A	•	•	•	•	•	•	00 000 010	1	2	7		
LD (DE), A	(DE) ← A	•	•	•	•	•	•	00 010 010	1	2	7		
LD (nn), A	(nn) ← A	•	•	•	•	•	•	00 110 010	3	4	13		
								← n →					
								← n →					
LD A, I	A ← I	•	‡	IFF	‡	0	0	11 101 101	2	2	9		
								01 010 111					
LD A, R	A ← R	•	‡	IFF	‡	0	0	11 101 101	2	2	9		
								01 011 111					
LD I, A	I ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 000 111					
LD R, A	R ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 001 111					

octal

Notes: r, r' means any of the registers A, B, C, D, E, H, L

IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

‡ = flag is affected according to the result of the operation.

16 bit loads

No flags affected!

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	•	•	•	•	00 dd0 001 ← n → ← n →	3	3	10	dd Pair 00 BC 01 DE 10 HL 11 SP
LD IX, nn	IX ← nn	•	•	•	•	•	•	11 011 101 00 100 001 ← n → ← n →	4	4	14	
LD IY, nn	IY ← nn	•	•	•	•	•	•	11 111 101 00 100 001 ← n → ← n →	4	4	14	
LD HL, (nn)	H ← (nn+1) L ← (nn)	•	•	•	•	•	•	00 101 010 ← n → ← n →	3	5	16	
LD dd, (nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	•	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	4	6	20	
LD IX, (nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	•	•	•	•	11 011 101 00 101 010 ← n → ← n →	4	6	20	
LD IY, (nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	•	•	•	•	11 111 010 00 101 010 ← n → ← n →	4	6	20	
LD (nn), HL	(nn+1) ← H (nn) ← L	•	•	•	•	•	•	00 100 010 ← n → ← n →	3	5	16	
LD (nn), dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	•	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	4	6	20	
LD (nn), IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	•	•	•	•	11 011 101 00 100 010 ← n → ← n →	4	6	20	
LD (nn), IY	(nn+1) ← IY_H (nn) ← IY_L	•	•	•	•	•	•	11 111 101 00 100 010 ← n → ← n →	4	6	20	
LD SP, HL	SP ← HL	•	•	•	•	•	•	11 111 001	1	1	6	
LD SP, IX	SP ← IX	•	•	•	•	•	•	11 011 101 11 111 001	2	2	10	
LD SP, IY	SP ← IY	•	•	•	•	•	•	11 111 101 11 111 001	2	2	10	
PUSH qq	(SP-2) ← qq_L (SP-1) ← qq_H	•	•	•	•	•	•	11 qq0 101	1	3	11	qq Pair 00 BC 01 DE 10 HL 11 AF
PUSH IX	(SP-2) ← IX_L (SP-1) ← IX_H	•	•	•	•	•	•	11 011 101 11 100 101	2	4	15	
PUSH IY	(SP-2) ← IY_L (SP-1) ← IY_H	•	•	•	•	•	•	11 111 101 11 100 101	2	4	15	
POP qq	qq_H ← (SP+1) qq_L ← (SP)	•	•	•	•	•	•	11 qq0 001	1	3	10	
POP IX	IX_H ← (SP+1) IX_L ← (SP)	•	•	•	•	•	•	11 011 101 11 100 001	2	4	14	
POP IY	IY_H ← (SP+1) IY_L ← (SP)	•	•	•	•	•	•	11 111 101 11 100 001	2	4	14	

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
E.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE ↔ HL	•	•	•	•	•	•	11 101 011	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	•	•	•	•	00 001 000	1	1	4	
EXX	$\begin{pmatrix}BC\\DE\\HL\end{pmatrix} \leftrightarrow \begin{pmatrix}BC'\\DE'\\HL'\end{pmatrix}$	•	•	•	•	•	•	11 011 001	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP+1) L ↔ (SP)	•	•	•	•	•	•	11 100 011	1	5	19	
EX (SP), IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	•	•	•	•	•	•	11 011 101 11 100 011	2	6	23	
EX (SP), IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	•	•	•	•	•	•	11 111 101 11 100 011	2	6	23	
LDI	(DE) ↔ (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 100 000	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	① 0	•	0	0	11 101 101 10 110 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 101 000	2	4	16	
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	① 0	•	0	0	11 101 101 10 111 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A - (HL) HL ← HL+1 BC ← BC - 1	•	② ↕	① ↕	↕	1	↕	11 101 101 10 100 001	2	4	16	
CPIR	A - (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 110 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A - (HL) HL ← HL-1 BC ← BC-1	•	② ↕	① ↕	↕	1	↕	11 101 101 10 101 001	2	4	16	
CPDR	A - (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 111 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD r	A ← A + r	↕	↕	V	↕	0	↕	10 [000] r	1	1	4	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
ADD n	A ← A + n	↕	↕	V	↕	0	↕	11 [000] 110 ← n →	2	2	7	
ADD (HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10 [000] 110	1	2	7	
ADD (IX+d)	A←A + (IX+d)	↕	↕	V	↕	0	↕	11 011 101 10 [000] 110 ← d →	3	5	19	
ADD (IY+d)	A←A+(IY+d)	↕	↕	V	↕	0	↕	11 111 101 10 [000] 110 ← d →	3	5	19	
ADC s	A ← A + s + CY	↕	↕	V	↕	0	↕	[001]				s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction
SUB s	A ← A − s	↕	↕	V	↕	1	↕	[010]				
SBC s	A ← A − s − CY	↕	↕	V	↕	1	↕	[011]				
AND s	A ← A ∧ s	0	↕	P	↕	1	↕	[100]				The indicated bits replace the 000 in the ADD set above.
OR s	A ← A ∨ s	0	↕	P	↕	0	↕	[110]				
XOR s	A ← A ⊕ s	0	↕	P	↕	0	↕	[101]				
CP s	A − s	↕	↕	V	↕	1	↕	[111]				
INC r	r ← r + 1	•	↕	V	↕	0	↕	00 r [100]	1	1	4	
INC (HL)	(HL) ← (HL)+1	•	↕	V	↕	0	↕	00 110 [100]	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	•	↕	V	↕	0	↕	11 011 101 00 110 [100] • d •	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d) + 1	•	↕	V	↕	0	↕	11 111 101 00 110 [100] ← d •	3	6	23	
DEC d	d ← d − 1	•	↕	V	↕	1	↕	[101]				d is any of r, (HL), (IX+d), (IY+d) as shown for INC. Same format and states as INC. Replace 100 with 101 in OP code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands	‡	‡	P	‡	•	‡	00 100 111	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	•	•	•	•	1	1	00 101 111	1	1	4	Complement accumulator (one's complement)
NEG	$A \leftarrow \overline{A} + 1$	‡	‡	V	‡	1	‡	11 101 101 01 000 100	2	2	8	Negate acc. (two's complement)
CCF	$CY \leftarrow \overline{CY}$	‡	•	•	•	0	X	00 111 111	1	1	4	Complement carry flag
SCF	$CY \leftarrow 1$	1	•	•	•	0	0	00 110 111	1	1	4	Set carry flag
NOP	No operation	•	•	•	•	•	•	00 000 000	1	1	4	
HALT	CPU halted	•	•	•	•	•	•	01 110 110	1	1	4	
DI	IFF ← 0	•	•	•	•	•	•	11 110 011	1	1	4	
EI	IFF ← 1	•	•	•	•	•	•	11 111 011	1	1	4	
IM 0	Set interrupt mode 0	•	•	•	•	•	•	11 101 101 01 000 110	2	2	8	
IM 1	Set interrupt mode 1	•	•	•	•	•	•	11 101 101 01 010 110	2	2	8	
IM2	Set interrupt mode 2	•	•	•	•	•	•	11 101 101 01 011 110	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

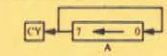
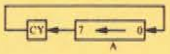
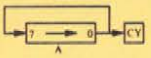
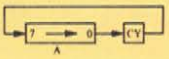
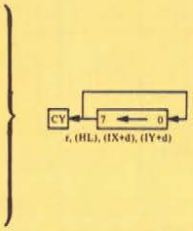
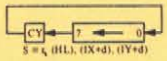
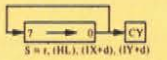
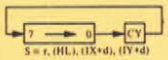
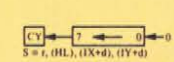
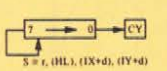
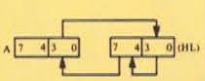
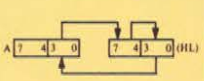
Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments	
		C	Z	P/V	S	N	H	76 543 210					
ADD HL, ss	HL ← HL+ss	‡	•	•	•	0	X	00 ss1 001	1	3	11	ss	Reg.
												00	BC
ADC HL, ss	HL←HL+ss+CY	‡	‡	V	‡	0	X	11 101 101	2	4	15	01	DE
								01 ss1 010				10	HL
SBC HL, ss	HL←HL-ss-CY	‡	‡	V	‡	1	X	11 101 101	2	4	15	11	SP
								01 ss0 010					
ADD IX, pp	IX ← IX + pp	‡	•	•	•	0	X	11 011 101	2	4	15	pp	Reg.
								00 pp1 001				00	BC
												01	DE
												10	IX
												11	SP
ADD IY, rr	IY←IY+rr	‡	•	•	•	0	X	11 111 101	2	4	15	rr	Reg.
								00 rr1 001				00	BC
												01	DE
												10	IY
												11	SP
INC ss	ss ← ss + 1	•	•	•	•	•	•	00 ss0 011	1	1	6		
INC IX	IX ← IX + 1	•	•	•	•	•	•	11 011 101	2	2	10		
								00 100 011					
INC IY	IY ← IY + 1	•	•	•	•	•	•	11 111 101	2	2	10		
								00 100 011					
DEC ss	ss ← ss - 1	•	•	•	•	•	•	00 ss1 011	1	1	6		
DEC IX	IX ← IX - 1	•	•	•	•	•	•	11 011 101	2	2	10		
								00 101 011					
DEC IY	IY ← IY - 1	•	•	•	•	•	•	11 111 101	2	2	10		
								00 101 011					

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
‡ = flag is affected according to the result of the operation.

8080 limited to these

Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA		‡	•	•	•	0	0	00 000 111	1	1	4	Rotate left circular accumulator
RLA		‡	•	•	•	0	0	00 010 111	1	1	4	Rotate left accumulator
RRCA		‡	•	•	•	0	0	00 001 111	1	1	4	Rotate right circular accumulator
RRA		‡	•	•	•	0	0	00 011 111	1	1	4	Rotate right accumulator
RLC r		‡	‡	P	‡	0	0	11 001 011 00 [000] r	2	2	8	Rotate left circular register r
RLC (HL)		‡	‡	P	‡	0	0	11 001 011 00 [000] 110	2	4	15	r Reg. 000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (IX+d)		‡	‡	P	‡	0	0	11 011 101 11 001 011 ← d → 00 [000] 110	4	6	23	
RLC (IY+d)		‡	‡	P	‡	0	0	11 111 101 11 001 011 ← d → 00 [000] 110	4	6	23	
RL s		‡	‡	P	‡	0	0	[010]				Instruction format and states are as shown for RLC,s. To form new OP-code replace [000] of RLC,s with shown code
RRC s		‡	‡	P	‡	0	0	[001]				
RR s		‡	‡	P	‡	0	0	[011]				
SLA s		‡	‡	P	‡	0	0	[100]				
SRA s		‡	‡	P	‡	0	0	[101]				
SRL s		‡	‡	P	‡	0	0	[111]				
RLD		•	‡	P	‡	0	0	11 101 101 01 101 111	2	5	18	Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected
RRD		•	‡	P	‡	0	0	11 101 101 01 100 111	2	5	18	

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
BIT b, r	$Z \leftarrow \overline{r}_b$	•	‡	X	X	0	1	11 001 011 01 b r	2	2	8	
BIT b, (HL)	$Z \leftarrow \overline{(HL)}_b$	•	‡	X	X	0	1	11 001 011 01 b 110	2	3	12	
BIT b, (IX+d)	$Z \leftarrow \overline{(IX+d)}_b$	•	‡	X	X	0	1	11 011 101 11 001 011 ← d → 01 b 110	4	5	20	
BIT b, (IY+d)	$Z \leftarrow \overline{(IY+d)}_b$	•	‡	X	X	0	1	11 111 101 11 001 011 ← d → 01 b 110	4	5	20	
SET b, r	$r_b \leftarrow 1$	•	•	•	•	•	•	11 001 011 [11] b r	2	2	8	
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	•	•	•	•	11 001 011 [11] b 110	2	4	15	
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	•	•	•	•	•	•	11 011 101 11 001 011 ← d → [11] b 110	4	6	23	
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	•	•	•	•	•	•	11 111 101 11 001 011 ← d → [11] b 110	4	6	23	
RES b, s	$s_b \leftarrow 0$ s ≡ r, (HL), (IX+d), (IY+d)							[10]				To form new OP-code replace [11] of SET b,s with [10]. Flags and time states for SET instruction

r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A

b	Bit Tested
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	•	•	•	•	•	•	11 000 011 ← n → ← n →	3	3	10	
JP cc, nn	If condition cc is true PC ←nn, otherwise continue	•	•	•	•	•	•	11 cc 010 ← n → ← n →	3	3	10	
JR e	PC ← PC + e	•	•	•	•	•	•	00 011 000 ← e-2 →	2	3	12	
JR C, e	If C = 0, continue	•	•	•	•	•	•	00 111 000 ← e-2 →	2	2	7	If condition not met
	If C = 1, PC ← PC+e								2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	•	•	•	•	00 110 000 ← e-2 →	2	2	7	If condition not met
	If C = 0, PC ← PC + e								2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	•	•	•	•	00 101 000 ← e-2 →	2	2	7	If condition not met
	If Z = 1, PC ← PC + e								2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	•	•	•	•	00 100 000 ← e-2 →	2	2	7	If condition not me
	If Z = 0, PC ← PC + e								2	3	12	If condition met
JP (HL)	PC ← HL	•	•	•	•	•	•	11 101 001	1	1	4	
JP (IX)	PC ← IX	•	•	•	•	•	•	11 011 101 11 101 001	2	2	8	
JP (IY)	PC ← IY	•	•	•	•	•	•	11 111 101 11 101 001	2	2	8	
DJNZ,e	B ← B-1 If B = 0, continue	•	•	•	•	•	•	00 010 000 ← e-2 →	2	2	8	If B = 0
	If B ≠ 0, PC ← PC + e								2	3	13	IF B ≠ 0

cc	Condition
000	NZ non zero
001	Z zero
010	NC non carry
011	C carry
100	PO parity odd
101	PE parity even
110	P sign positive
111	M sign negative

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <-126, 129>

e-2 in the op-code provides an effective address of pc +e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	$(SP-1)\leftarrow PC_H$ $(SP-2)\leftarrow PC_L$ $PC\leftarrow nn$	•	•	•	•	•	•	11 001 101 ← n → ← n →	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	•	•	•	•	11 cc 100 ← n → ← n →	3 3	3 5	10 17	If cc is false If cc is true
RET	$PC_L\leftarrow(SP)$ $PC_H\leftarrow(SP+1)$	•	•	•	•	•	•	11 001 001	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	•	•	•	•	11 cc 000	1 1	1 3	5 11	If cc is false If cc is true
RETI	Return from interrupt	•	•	•	•	•	•	11 101 101 01 001 101	2	4	14	
RETN	Return from non maskable interrupt	•	•	•	•	•	•	11 101 101 01 000 101	2	4	14	
RST p	$(SP-1)\leftarrow PC_H$ $(SP-2)\leftarrow PC_L$ $PC_H\leftarrow 0$ $PC_L\leftarrow P$	•	•	•	•	•	•	11 t 111	1	3	11	

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	P
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown
‡ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
IN A, n	A ← (n)	•	•	•	•	•	•	11 011 011 ← n →	2	3	10	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	•	‡	P	‡	0	‡	11 101 101 01 r 000	2	3	11	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INI	(HL) ← (C) B ← B - 1 HL ← HL + 1	•	‡ ①	X	X	1	X	11 101 101 10 100 010	2	4	15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INIR	(HL) ← (C) B ← B - 1 HL ← HL + 1 Repeat until B = 0	•	1	X	X	1	X	11 101 101 10 110 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	•	‡ ①	X	X	1	X	11 101 101 10 101 010	2	4	15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
INDR	(HL) ← (C) B ← B - 1 HL ← HL - 1 Repeat until B = 0	•	1	X	X	1	X	11 101 101 10 111 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUT n, A	(n) ← A	•	•	•	•	•	•	11 010 011	2	3	11	n to $A_0 \sim A_7$ Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	•	•	•	•	11 101 101 01 r 001	2	3	12	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	•	‡ ①	X	X	1	X	11 101 101 10 100 011	2	4	15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 Repeat until B = 0	•	1	X	X	1	X	11 101 101 10 110 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	•	‡ ①	X	X	1	X	11 101 101 10 101 011	2	4	15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL - 1 Repeat until B = 0	•	1	X	X	1	X	11 101 101 10 111 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	20 15	C to $A_0 \sim A_7$ B to $A_8 \sim A_{15}$

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Appendix C
UNDOCUMENTED Z80 OPCODES

Every microprocessor has some instructions which may work one way on a given chip and quite another way on the next chip off the production line. Although it may be fun to discover these things, I suggest you avoid it altogether. Even if you find a neat instruction which works on your machine, and even though you have no plans to write programs for commercial marketing, you may still be thwarted by something as simple as a trip to the repair shop -- any time the Z80 chip in your machine needs replacement, your programs using those special instructions probably will not work in the same way now. So stay out of this mess, unless the "thrill of the chase" yields enough pleasure to compensate for the inevitable frustration.

This type of "phantom" instruction is not the subject of this chapter! There is a whole set of Z80 instructions which (apparently) are sure to work on every Z80 chip, but have not been publicly documented by Zilog in any of their publications. Although I do not know the reasons for this secrecy, it makes me somewhat uneasy. Prudence forces me to recommend that you check any and all instructions I am about to reveal to you to ascertain that they indeed perform as I describe.

The principal sources of my information about these hidden instructions are two excellent articles which appeared in The Alternate Source (TAS). TAS is a magazine (they are now calling it a "Programmer's Journal") to which you should seriously consider subscribing. The pertinent address information is:

The Alternate Source
704 North Pennsylvania Ave.
Lansing, MI 48906
(517) 482-8270

The specific article references are shown at the end of the text of this chapter. I will use the same mnemonics as shown in the TAS articles to minimize confusion.

The new instructions fall into two main groups:

1. A "shift left and increment" instruction for all general-purpose 8-bit registers and memory pointed to by HL, IX, or IY.

2. A whole set of instructions involving only one of the bytes of IX or IY.

The first group uses the mnemonic "SLS" for "shift left one bit and set the least significant bit (bit 0)". The shift out of bit 7 goes into the carry bit, and sign and parity flags are affected in the expected ways. The net effect of this instruction is a multiplication by 2 and then addition of 1 to the result. The instructions take four clock cycles longer than the corresponding "SLA" instructions on page B-8.

The second group treats the high and low bytes of the index registers separately. In other words, the heretofore "inseparable" index registers can be treated as composed of two 8-bit registers, analogous to the HL register pair. The high byte of IX is called "HX", the low byte is "LX"; for IY, the high byte is "HY" and the low byte is "LY". The machine codes for these instructions are formed by placing a "prefix" byte before the analogous instruction involving the H or L register. This prefix byte is 0DDH for operations involving IX and 0FDH for operations with IY.

For example, to implement

```
ADC   A,HX
```

we could use the following two-instruction sequence:

```
DEFB  0DDH
ADC   A,H
```

Likewise, for the instruction:

```
ADC   A,LY
```

we could substitute:

```
DEFB  0FDH
ADC   A,L
```

This pattern should be obvious as you study the table further. Note that neither the H nor the L register can be loaded to/from the "new" registers. Also, instructions such as "LD LX,LY" are not implemented. The timings of all of these new instructions are predictable, since they take four clock cycles more than the analogous instructions for H or L, as shown in Appendix B.

All of the new instructions, in alphabetical order and accompanied by the hex codes, are given beginning on the next page.

TAS REFERENCES

Daniel R. Lunsford, "Undocumented Z-80 Opcodes", TAS Vol. I, No. 6, page 53.

Dennis Bathory Kitsz, "A Manual for Radio Shack's Editor Assembler, Part Three", TAS Issue 13, page 3.

TABLE OF UNDOCUMENTED OPCODES

KEY: d = one-byte expression -128 to +127
n = one-byte expression 0 to +255
r = A, B, C, D, E, H, L

MNEMONIC		SUGGESTED OPCODE		
ADC	A,HX	DEFB 0DDH +	ADC	A,H
ADC	A,HY	DEFB 0FDH +	ADC	A,H
ADC	A,LX	DEFB 0DDH +	ADC	A,L
ADC	A,LY	DEFB 0FDH +	ADC	A,L
ADD	A,HX	DEFB 0DDH +	ADD	A,H
ADD	A,HY	DEFB 0FDH +	ADD	A,H
ADD	A,LX	DEFB 0DDH +	ADD	A,L
ADD	A,LY	DEFB 0FDH +	ADD	A,L
AND	HX	DEFB 0DDH +	AND	H
AND	HY	DEFB 0FDH +	AND	H
AND	LX	DEFB 0DDH +	AND	L
AND	LY	DEFB 0FDH +	AND	L
CP	HX	DEFB 0DDH +	CP	H
CP	HY	DEFB 0FDH +	CP	H
CP	LX	DEFB 0DDH +	CP	L
CP	LY	DEFB 0FDH +	CP	L
DEC	HX	DEFB 0DDH +	DEC	H
DEC	HY	DEFB 0FDH +	DEC	H
DEC	LX	DEFB 0DDH +	DEC	L
DEC	LY	DEFB 0FDH +	DEC	L
INC	HX	DEFB 0DDH +	INC	H
INC	HY	DEFB 0FDH +	INC	H
INC	LX	DEFB 0DDH +	INC	L
INC	LY	DEFB 0FDH +	INC	L
LD	A,HX	DEFB 0DDH +	LD	A,H
LD	A,HY	DEFB 0FDH +	LD	A,H
LD	A,LX	DEFB 0DDH +	LD	A,L
LD	A,LY	DEFB 0FDH +	LD	A,L
LD	B,HX	DEFB 0DDH +	LD	B,H
LD	B,HY	DEFB 0FDH +	LD	B,H
LD	B,LX	DEFB 0DDH +	LD	B,L
LD	B,LY	DEFB 0FDH +	LD	B,L
LD	C,HX	DEFB 0DDH +	LD	C,H
LD	C,HY	DEFB 0FDH +	LD	C,H
LD	C,LX	DEFB 0DDH +	LD	C,L

LD	C,LY	DEFB 0FDH + LD C,L
LD	D,HX	DEFB 0DDH + LD D,H
LD	D,HY	DEFB 0FDH + LD D,H
LD	D,LX	DEFB 0DDH + LD D,L
LD	D,LY	DEFB 0FDH + LD D,L
LD	E,HX	DEFB 0DDH + LD E,H
LD	E,HY	DEFB 0FDH + LD E,H
LD	E,LX	DEFB 0DDH + LD E,L
LD	E,LY	DEFB 0FDH + LD E,L
LD	HX,LX	DEFB 0DDH + LD H,L
LD	HX,n	DEFB 0DDH + LD H,n
LD	HX,r	DEFB 0DDH + LD H,r (except H or L)
LD	HY,LY	DEFB 0FDH + LD H,L
LD	HY,n	DEFB 0FDH + LD H,n
LD	HY,r	DEFB 0FDH + LD H,r (except H or L)
LD	LX,HX	DEFB 0DDH + LD L,H
LD	LX,n	DEFB 0DDH + LD L,n
LD	LX,r	DEFB 0DDH + LD L,r (except H or L)
LD	LY,HY	DEFB 0FDH + LD L,H
LD	LY,n	DEFB 0FDH + LD L,n
LD	LY,r	DEFB 0FDH + LD L,r (except H or L)
OR	HX	DEFB 0DDH + OR H
OR	HY	DEFB 0FDH + OR H
OR	LX	DEFB 0DDH + OR L
OR	LY	DEFB 0FDH + OR L
SBC	A,HX	DEFB 0DDH + SBC A,H
SBC	A,HY	DEFB 0FDH + SBC A,H
SBC	A,LX	DEFB 0DDH + SBC A,L
SBC	A,LY	DEFB 0FDH + SBC A,L
SLS	(HL)	DEFB 0CBH, 36H
SLS	(IX+d)	DEFB 0DDH, 0CBH, d, 36H
SLS	(IY+d)	DEFB 0FDH, 0CBH, d, 36H
SLS	A	DEFB 0CBH, 37H
SLS	B	DEFB 0CBH, 30H
SLS	C	DEFB 0CBH, 31H
SLS	D	DEFB 0CBH, 32H
SLS	E	DEFB 0CBH, 33H
SLS	H	DEFB 0CBH, 34H
SLS	L	DEFB 0CBH, 35H
SUB	HX	DEFB 0DDH + SUB H
SUB	HY	DEFB 0FDH + SUB H
SUB	LX	DEFB 0DDH + SUB L
SUB	LY	DEFB 0FDH + SUB L
XOR	HX	DEFB 0DDH + XOR H
XOR	HY	DEFB 0FDH + XOR H
XOR	LX	DEFB 0DDH + XOR L
XOR	LY	DEFB 0FDH + XOR L

FROM_____

_____

_____ZIP #_____

Place
Stamp
Here

REMSOFT, INC.
571 East 185th Street
Euclid, Ohio 44119

REMSOFT, INC.

571 East 185th Street
Euclid, Ohio 44119

Use this form for questions on assembly language programming which are within the scope of the RAMESSEM 1 course. Do not use this form for any other purpose as it will be opened by a programmer, not the business office. Be sure to fill in your name below as this will be the address your reply will be mailed

Name _____

Address _____

City / State / Zip _____

Subject _____ Date _____

Signed _____

Reply _____ Date _____

Signed _____

staple
or tape here

fold line

FROM_____

_____

_____ZIP #_____

Place
Stamp
Here

REMSOFT, INC.
571 East 185th Street
Euclid, Ohio 44119

fold line

REMSOFT, INC.

571 East 185th Street
Euclid, Ohio 44119

Use this form for questions on assembly language programming which are within the scope of the RAMESSEM 1 course. Do not use this form for any other purpose as it will be opened by a programmer, not the business office. Be sure to fill in your name below as this will be the address your reply will be mailed

Name _____

Address _____

City / State / Zip _____

Subject _____ Date _____

Signed _____

Reply _____ Date _____

Signed _____